HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Entwicklung einer abgesicherten Ausführungsumgebung für automatisierte Bewertungsprozesse

Kai Dierschke

Bachelor-Arbeit im Studiengang „Angewandte Informatik“

30. November 2018

Autor Kai Dierschke
1381261
kai.die@web.de

Erstprüfer: Prof. Dr. Robert Garmann
Abteilung Informatik, Fakultät IV
Hochschule Hannover
robert.garmann@hs-hannover.de

Zweitprüfer: Peter Fricke
ZLB - E-Learning Center (ELC)
Hochschule Hannover
peter.fricke@hs-hannover.de

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 30. November 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Abkürzungen und Begriffe

Lernmanagementsystem: Ein im Bereich der Lehre eingesetztes System, mit welchem Lerninhalte bereitgestellt sowie Lernvorgänge und Benutzer organisiert werden können.

LMS: Synonym für Lernmanagementsystem

Autobewerter: Ein im Bereich der Lehre eingesetztes System, mit welchem eingereichte Lösungen zu (Programmier-) Aufgaben automatisiert oder teilautomatisiert bewertet werden können.

Grader: Synonym für Autobewerter

Kernel: Zentrales Bestandteil eines Betriebssystems. Ein Kernel hat direkten Zugriff auf die Hardware und andere Softwarekomponenten des Betriebssystems bauen auf diesen auf.

Systemaufruf: Eine von Software aufrufbare Methode, um vom Betriebssystem bereitgestellte Funktionalitäten auszuführen.

Kontinuierliche Integration: Prozess, welcher das fortlaufende Zusammenfügen von Softwarekomponenten zu einer Anwendung ermöglicht.

2. Einführung

In der Lehre kommen seit einiger Zeit Autobewerter zum Einsatz, um automatisiert oder teilautomatisiert Lösungen von Lernenden zu Programmieraufgaben zu bewerten. Aus Sicht eines Betreibers von Autobewerter-Systemen erfordert letzteres meist die Ausführung von fremdem Programmcode in der eigenen Server-Umgebung. Hierbei ist Vorsicht geboten, da fehlerhafter oder angriffslustiger Programmcode verbotene Aktionen ausführen könnte.
An der Hochschule Hannover können Studierende Lösungen zu Programmieraufgaben verschiedener Programmiersprachen über das Lernmanagementsystem „moodle“ einreichen. Die angeschlossene Middleware Grappa nimmt Lösungen anschließend entgegen und leitet diese an einen Autobewerter der zugehörigen Programmiersprache weiter.

Diese Arbeit befasst sich damit, Sicherheitsrisiken bei der Ausführung von fremdem Programmcode durch Autobewerter zu identifizieren und zu vermeiden. Hierzu gilt es, mögliche Angriffe herauszustellen und aktuelle, angemessene Technologien zu finden, um diese zu verhindern. Anschließend findet mit einer Technologie die Entwicklung einer abgesicherten Ausführungsumgebung für beliebige an die Middleware Grappa angebundene Autobewerter statt. Die exemplarische Einbettung der beiden Autobewerter Graja und Praktomat in die entwickelte Ausführungsumgebung soll unter Beweis stellen, dass es sich um eine generische Lösung mit wenig Overhead handelt.

Andere Autobewerter-Systeme nutzen bereits entweder Eigenentwicklungen ([MB12]), virtuelle Maschinen ([Lin14]) oder Container ([GBSO17], [ŠSD15]), um eine abgesicherte Ausführungsumgebung bereitzustellen. Insbesondere diese Lösungsmöglichkeiten werden im Rahmen dieser Arbeit bzgl. Sicherheit und Effizienz untersucht.

2.1. Herangehensweise und Leseanleitung

Die in Kapitel 3.3 vorgestellten Angriffsszenarien entstanden mit Hilfe von Literatur aus den dort genannten Quellen und anschließender Auswertung.
Aus den möglichen Angriffen wurden anschließend Gegenmaßnahmen abgeleitet, welche dann Einzug in die vorgestellten Anforderungen erhielten (siehe Kapitel 4.1). Weitere Anforderungen entstanden aus Dialogen mit den Systemverantwortlichen von Grappa und dessen Umfeld (= die Betreuer dieser Arbeit). Weiterhin wurden mit Hilfe der Anforderungen allgemeingültige Use-Cases für spätere konkrete Tests formuliert.

In Kapitel 5 hat anschließend eine Evaluation der Lösungsmöglichkeiten und verwendbaren Technologien zur Umsetzung aller Anforderungen stattgefunden. Hierbei wurde sich auf Technologiedokumentationen, Erfahrungswerte von Entwicklern ähnlicher Systeme sowie Best Practices gestützt.
Die am besten geeignete Lösungsmöglichkeit wurde schließlich ausgewählt und ein Konzept zur konkreten Implementierung entwickelt und umgesetzt (siehe Kapitel 6).
Abschließend ist die implementierte Lösung bzgl. Funkion, Effizienz und Sicherheit geprüft worden. Hierzu waren u. a. die zuvor definierten Use-Cases von Nöten. Die Testergebnisse und eine zugehörige Auswertung sind in Kapitel 7 zu finden.

3. Ist-Zustand

In diesem Kapitel wird erläutert, wie die Middleware Grappa sowie dessen Umfeld vorgefunden wurde. Außerdem werden die für diese Arbeit relevaten Abläufe erklärt und mögliche Angriffe auf Autobewerter herausgestellt.

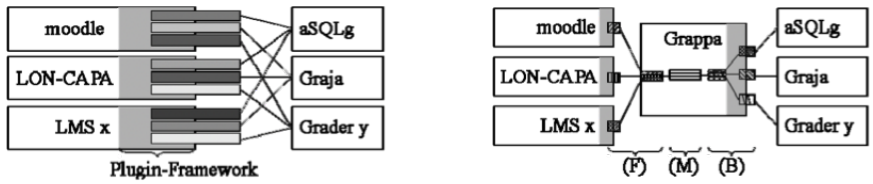
3.1. Beschreibung von Grappa und dessen Umfeld

Grappa ist eine an der Hochschule Hannover entwickelte und sich im produktiven Einsatz befindende Middleware. Ihr Einsatzgebiet liegt im LMS- und Grader-Umfeld und sie wird auch als „Spinne im Netz der Autobewerter und Lernmanagementsysteme“ bezeichnet (vgl. [GHW15]).
Grappa wurde mit dem Ziel entwickelt, eine standardisierte Schnittstelle zwischen beliebigen LMSen und Gradern zu schaffen, um so den Aufwand zu minimieren, welcher bei der Anbindung eines Graders an ein LMS entsteht. Prinzipiell ist es zwar möglich, Grader direkt an LMSe (über LMS-Plugin-Frameworks) anzubinden, jedoch führt dies dazu, dass auf Dauer für jedes LMS-Grader-Paar eine proprietäre Schnittstelle entwickelt und gepflegt werden müsste (siehe Abb. 3.1a). Grappa ist LMS- und graderunabhängig, sodass für jedes LMS und jeden Grader nur eine einzige Schnittstelle entwickelt werden muss (siehe Abb. 3.1b).
Um die technische Anbindung von LMSen und Gradern, soweit es geht, zu vereinfachen, bietet Grappa in beiden Richtungen schlanke Schnittstellen in Form von Bibliotheken/Plugins an und übernimmt intern einige LMS-übergreifende Aufgaben. Um dies zu realisieren, besteht Grappa's Architektur aus:

Einer Frontend-Komponente (F), welche den LMSen (bzw. LMS-Plugins) Clientbibliotheken in verschiedenen Programmiersprachen anbietet. Mit Hilfe solch einer Bibliothek ist es aus einem LMS möglich, Konfigurationen und Programmieraufgaben zu hinterlegen sowie den Bewertungsprozess asynchron anzusteuern.

Einer Backend-Komponente (B), welche über ein Backend-Plugin einen Grader ansteuert. In einem Backend-Plugin ist die technische Anbindung an einen Grader gekapselt.

Einer Kern-Komponente (M). Hier geschieht das Queueing von Anfragen, die Umwandlung von Bewertungsskalen sowie ggf. der Austausch von Programmieraufgaben über LMS-Grenzen hinweg.



(a) Direkte Anbindung von LMS und Grader

(b) Anbindung von LMS und Grader über Grappa

Abbildung 3.1.: Anbindungsarten von LMSen und Gradern (Quelle: [GHW15])

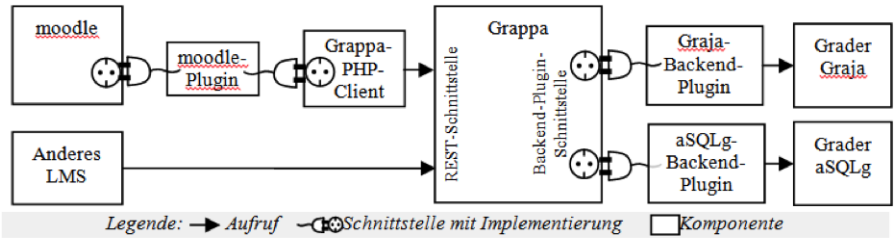


Abbildung 3.2.: Grappa - Architektur (Quelle: [FGH+15])

In Abbildung 3.2 ist Grappa's Architektur mit beispielhaften Anbindungen an das LMS „moodle“ sowie an die Grader „Graja“ und „aSQLg“ dargestellt.
Weitere Spezifikationen sind in den zugehörigen Quellen [GHW15], [BFPS17], [FGH+15] zu finden.

Technisch wurde Grappa in der Programmiersprache Java umgesetzt und muss für jeden anzubindenden Grader als eigenständiger Webservice in einen Application-Server deployt werden. Für jeden dieser Webservices ist zudem eine Konfiguration zu hinterlegen, in welcher das zu verwendende Grader-Plugin und eine Datenbankverbindung definiert sind (siehe die dem Grappa-Quellcode beiliegende Installionsanleitung).

Der Grader Graja

Bei Graja handelt es sich um einen Grader für Java-Programme. Dieser wurde an der Hochschule Hannover entwickelt und befindet sich dort seit 2012 im produktiven Einsatz.
Bei der Bewertung von Abgaben stützt sich Graja auf den Java Compiler, dynamische Softwaretests und statische Analysen des Quellcodes. Ein Aufgabenautor kann eine von Graja bewertbare Aufgabenstellung erstellen, indem er ein zip-Archiv mit einer vorgegebenen Struktur erstellt. In diesem Archiv sind Dateien enthalten, welche die Aufgabenbeschreibung, Komponenten zur Lösungsüberprüfung, Musterlösungen und Konfigurationen der Graderumgebung definieren.
Graja wurde in Java implementiert und ist als autonomes Programm über die Kommandozeile ausführbar. Zudem existiert eine Java-API, welche u. a. dazu benutzt wurde, Gra-

ja's Anbindung an Grappa zu realisieren. Um Graja in Grappa funktionsfähig zu machen sind Java, eine Graja-Installation sowie das Einkonfigurieren des Graja-Backend-Plugins erforderlich. (Siehe [BFPS17], [Gar18])

Der Grader Praktomat

Beim Praktomaten handelt es sich um einen Grader für diverse Programmiersprachen. Der Praktomat ermöglicht von Haus aus Code-Abgaben in Java, C, C++, Fortran, Haskell, R oder Isabelle zu bewerten. Die Unterstützung für weitere Programmiersprachen und Bewertungsarten kann durch das Hinzufügen von sogenannten „Checker"-Modulen ergänzt werden. Neben der reinen Bewertungsfunktionalität bietet der Praktomat zusätzlich eine Weboberfläche mit LMS-ähnlichen Funktionalitäten, wie das Erstellen von Programmieraufgaben oder das direkte Einreichen einer Abgabe.
Der Praktomat ist als webbasierter Dienst in Python mit dem Webframework „Django" implementiert. Im Rahmen einer Bachelorarbeit (vgl. [Tos18]) wurde die Bewertungsfunkionalität des Praktomats an Grappa über eine HTTP-Schnittstelle angebunden und ein zusätzlicher Checker entwickelt. Ein Aufgabenautor kann nun außerdem eine Aufgabenstellung in Form einer xml-Datei über das LMS „moodle" hinterlegen. Aus Grappa heraus kann so die Bewertung von Python-Code erfolgen. Um den Praktomat in Grappa funktionsfähig zu machen, sind Python, eine virtuelle Python-Umgebung, eine Postgres-Datenbank, ein Apache-Webserver, die Praktomat-Dateien sowie das Einkonfigurieren des Praktomat-Backend-Plugins erforderlich. (Siehe [BFPS17], [Tos18])

3.2. Bewertungsablauf

Der grundsätzliche Bewertungsablauf mit Grappa ist in Abbildung 3.3 verkürzt dargestellt. Im Folgenden werden Prozesse und Komponeten, welche Teil des LMSs, Grappa's Frontend-Komponente oder Grappa's allgemeiner Verarbeitung sind, nur oberflächlich behandelt. Hauptaugenmerk sind Prozesse und Komponeten, welche im direktem Zusammenhang mit dem Anstoß einer Bewertung durch den Grader und dem Zurückliefern des Bewertungsergebnisses an Grappa stehen. Eine umfassendere Beschreibung ist in [BFPS17] zu finden.

Vorbedingungen

Bevor die Abgabe einer Programmieraufgabe über ein LMS überhaupt möglich wird, müssen zunächst sogenannte „GrdCfgs" in Grappa bzw. in der Grappa-Datenbank hinterlegt werden. Hierbei handelt es sich um graderspezifische Konfigurationsdateien. Des

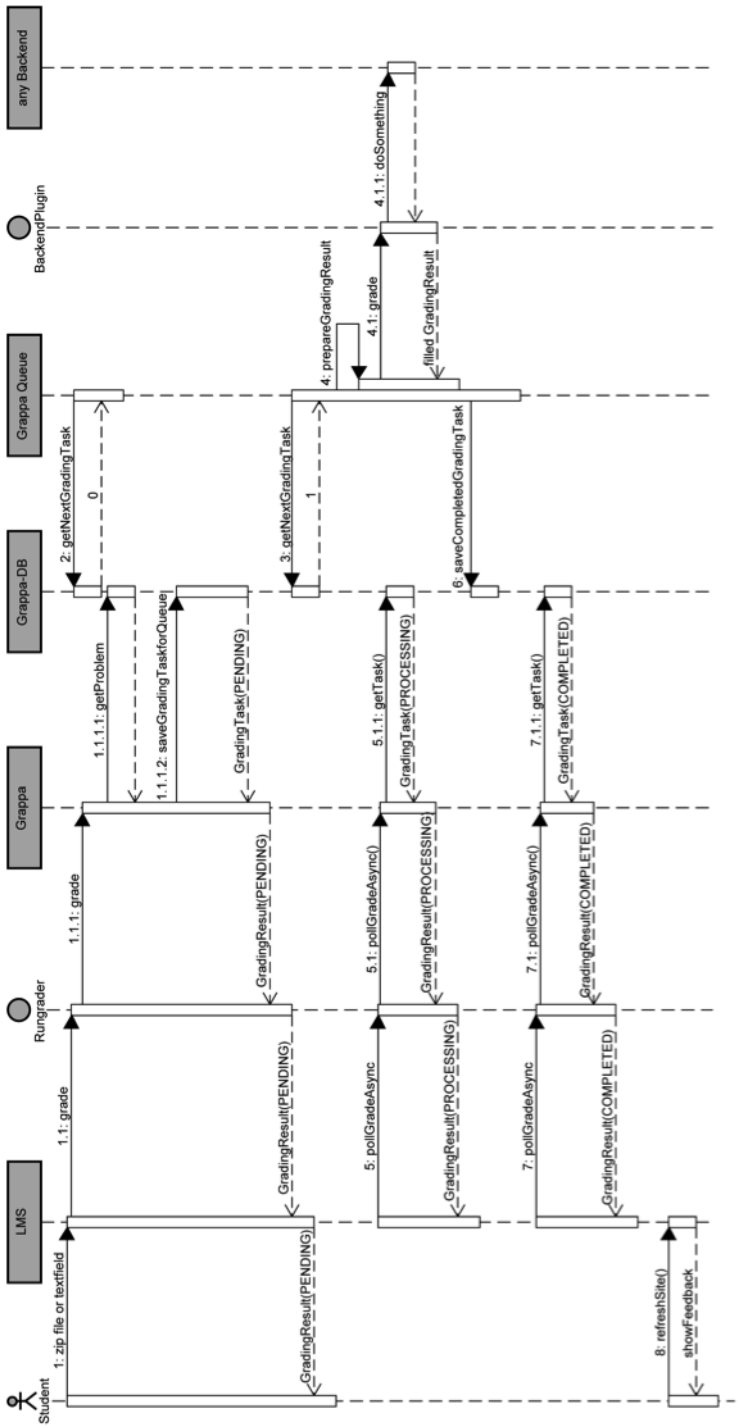


Abbildung 3.3.: Bewertungsablauf in Grappa (Quelle: [BFPS17])

Weiteren muss ein „Problem“ in Grappa bzw. in der Grappa-Datenbank angelegt werden. GrdCfgs und Problems tragen dann dazu bei, eine konkrete Programmieraufgabe zu definieren. Im Praxisbetrieb wird beides von einem Lehrenden über das LMS erledigt.

Abgabe im LMS

Die Abgabe einer Lösung kann im LMS sowohl als (zip-) Datei, als auch in einem Textfeld erfolgen. Das zugehörige LMS-Plugin leitet die Abgabe dann zusammen mit den notwendigen Problem-Informationen an Grappa zur Verarbeitung weiter. Das LMS führt nun in regelmäßigen Zeitintervallen ein Polling durch, bis Grappa ein Bewertungsergebnis zurückliefert.

Verarbeitung in Grappa

Nachdem eine Abgabe in Grappa eingetroffen ist, wird aus dieser und den zugehörigen Problem-Daten zunächst ein „GradingTask“ konstruiert und in der Datenbank persistiert. Der GradingTask ist nun in einer Queue eingereiht und kann dieser entnommen und bearbeitet werden, sofern Ressourcen frei sind.

Anstoßen der Bewertung

Ist ein GradingTask aus der Queue entnommen, wird dieser bearbeitet, indem der Aufruf eines Backend-Plugins stattfindet. Im Backend-Plugin erfolgt dann die Übermittlung nötiger Daten an einen Grader, damit dieser wiederum die Bewertung durchführen kann. Abbildung 3.4 zeigt die konkrete Definition der „BackendPlugin“-Schnittstelle. Die „grade“-Methode ist dafür verantwortlich, dass die Abgabe inklusive aller zur Bewertung benötigten Daten in Form eines „Submission“-Objekts bereitgestellt werden und der Grader ausgeführt wird. Das übergebene „GradingResult“-Objekt ist anschließend mit dem Bewertungsergebnis befüllt, welches sich in einem von Grappa akzeptierten und vom LMS angeforderten Format befindet.
Die Art und Weise, wie Grader innerhalb eines BackendPlugins aufgerufen werden, ist beliebig und abhängig von den angebotenen Schnittstellen eines Graders. Es kann sich beispielsweise um eine Java-Bibliothek, ein Kommandozeilentool oder einen Webservice handeln.

Zurückliefern des Ergebnisses

Sobald das befüllte GradingResult vorliegt, wird neben dem zugehörigen GradingTask auch dieses in der Datenbank persistiert und kann beim nächsten Polling durch das LMS abgeholt werden. Das LMS kann dem Nutzer nun das Feedback des Graders zu seiner Abgabe anzeigen.

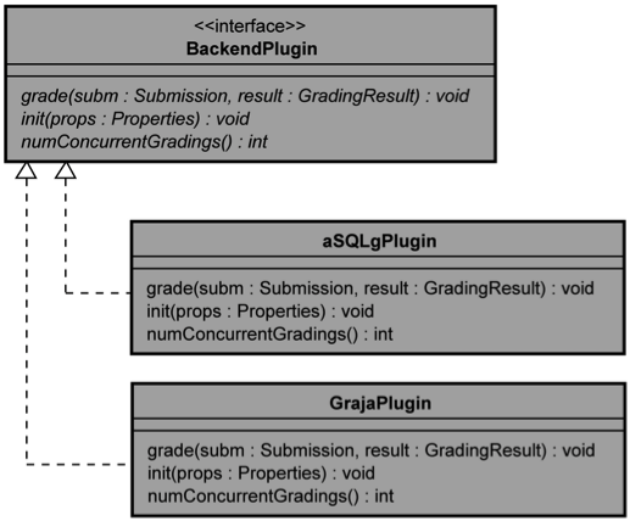


Abbildung 3.4.: BackendPlugin-Schnittstelle (Quelle: [BFPS17])

3.3. Mögliche Angriffsszenarien

Im Folgenden werden mögliche Angriffe auf Autobewerter-Systeme vorgestellt. Diese sind das Ergebnis einer Analyse, basierend auf [For06], [TB10], [MB12], [Lin14], [GBSO17], [Gar13] und eine Sammlung verschiedener bekannter Angriffe. Eine durch Beweisführung erstellte, allumfassende Liste mit bisher unbekannten oder erst in Zukunft hervorbringbaren Angriffen, ist aus offensichtlichen Gründen im Rahmen dieser Arbeit nicht möglich. Die Angriffe wurden kategorisiert und sind unabhängig von der vom Autobewerter verwendeten Programmiersprache und dessen Möglichkeiten definiert.

Risiken beim Initialisieren/Kompilieren einer Abgabe

- Bereits bei der Initialisierung einer Abgabe werden häufig Archive entpackt und Inhalte kopiert. Hierbei kann es bei speziell präparierten Abgaben dazu kommen, dass Dateien in nicht vorhersehbare Pfade geschrieben werden. Möglich wird dies beispielsweise durch Pfadtraversierung oder Symlinks.
- Weiterhin kann das Einreichen einer sehr großen Abgabe oder einer Archivbombe lange Zeiten beim Entpacken oder das Erschöpfen der Arbeitsspeicher-/Festplattenkapazitäten hervorrufen und damit zu einer Blockade oder Verlangsamung des Systems führen.
- Der eingereichte Quellcode kann Anweisungen beinhalten, welche den Kompilierprozess niemals enden lassen oder Systemressourcen stark beanspruchen. Dies kann ebenfalls eine Blockade oder Verlangsamung des Systems verursachen.

- Bereits beim Kompilieren kann ein Inkludieren sensibler Daten oder Musterlösungen erfolgen, sofern der Kompilierprozess Zugriff auf diese hat. Die sensiblen Daten oder Fremdlösungen könnten anschließend von einem Angreifer bei der Bewertung/Ausführung nutzbar gemacht werden.

Übermäßige Beanspruchung von Systemressourcen beim Ausführen einer Abgabe

Eine Abgabe kann zur Laufzeit verschiedene Systemressourcen dauerhaft stark beanspruchen. Hierbei muss es sich nicht zwingend um einen Angriff handeln, da ein simpler Programmierfehler dasselbe verursachen kann. In beiden Fällen ist das Resultat eine Blockade oder Verlangsamung des Systems bzw. von Komponenten. Solche Angriffe werden auch als DoS (Denial of Service)-Attacken bezeichnet.

- Eine Beanspruchung der CPU oder übermäßig lange Rechenzeiten können durch aktives Warten, Suspendieren eigener Prozesse/Threads oder Erzeugen von weiterer Prozessen/Threads entstehen.
- Werden im eingereichten Code Betriebssystemressourcen beansprucht aber nicht wieder freigegeben, kann dies zu einer Blockade des Systems führen. Beispiele hierfür sind nicht geschlossene Dateideskriptoren/Streams oder aufgebrauchte Prozess-/Thread-Pools.
- Eine Erschöpfung und Beanspruchung des Arbeitsspeichers kann durch übermäßige Allokation verursacht werden.
- Eine Erschöpfung des Festplattenspeichers oder übermäßiger I/O-Zugriff kann durch Schreiben großer Dateien auf die Festplatte bzw. häufiges Durchschreiben von Puffern auf die Festplatte erfolgen.
- Versenden von vielen Daten und Anfragen über die Netzwerkschnittstelle können zu einer Verlangsamung oder einem Ausfall dieser und ggf. des gesamten Netzwerkes führen.
- Werden sehr viele Ausgaben an die Standardausgabe, Standardfehlerausgabe oder an eine Log-Datei weitergeleitet, ist es möglich, dass das System selbst oder eine für die Weiterverarbeitung zuständige Komponente blockiert wird oder weitere Ressourcen erschöpft werden.

System beim Ausführen einer Abgabe in einen inkonsistenten Zustand bringen

- Sobald eingereichter Code schreibende Zugriffsrechte im Dateisystem außerhalb des eigenen Arbeitsbereiches hat, ist es dem Code möglich, ggf. wichtige Dateien zu überschreiben, zu löschen, zu erzeugen oder Berechtigungen zu ändern. Dies kann dazu führen, dass das System in einen fehlerhaften Zustand gebracht wird. Weiterhin ist es möglich, dass Änderungen an Konfigurationen vorgenommen oder

ausführbare Dateien des Systems ausgetauscht werden, damit ein Angreifer neben dem Ausführen seiner Abgabe, erweiterten Zugriff auf das System erlangt. Sollte der eingereichte Code ebenfalls schreibende Zugriffe auf Bewertungsergebnisse im Dateisystem besitzen, ist es einem Angreifer möglich auch diese zu ändern.

- Besitzt eingereichter Code bestimmte ausführende Rechte, ist es einem Angreifer ggf. möglich, Systemzustände und -konfigurationen zu ändern, Rechte auszuweiten oder zu einem Benutzer mit erweiterten Privilegien zu wechseln.
- Ein Angreifer könnte durch das Senden von Signalen anderen Komponenten Befehle mitteilen. Mit Hilfe von Signalen ist es u. a. möglich, andere Prozesse zu terminieren, Abläufe durcheinander zu bringen oder gar das gesamte System herunterzufahren.
- Sofern eingereichter Code schreibenden Zugriff auf sensible Teile des Hauptspeichers hat, ist es einem Angreifer ggf. möglich, in Abläufe oder in die Bewertung einzugreifen.
- Auch wenn die Möglichkeit besteht, Nachrichten über Interprozesskommunikation an fremde Komponenten zu senden, ist es einem Angreifer ggf. möglich, in Abläufe oder in die Bewertung einzugreifen.
- Durch das Ausnutzen von Sicherheitslücken im Betriebssystem oder in anderen Softwarekomponenten kann es einem Angreifer durch Privilegien-Eskalation möglich werden, umfassenden Zugriff auf das System zu erlangen.

Ausspionieren von Informationen beim Ausführen einer Abgabe

- Sobald eingereichter Code lesende Zugriffsrechte im Dateisystem außerhalb des eigenen Arbeitsbereiches hat, ist es dem Code möglich, ggf. sensible Informationen und Geheimnisse auszulesen. Es kann sich beispielsweise um IP-Adressen oder Login-Daten handeln, welche in Konfigurationsdateien hinterlegt sind oder in Log-Dateien auftauchen. Auch das Auslesen von Musterlösungen wäre möglich, falls sich diese im Dateisystem befinden.
- Besitzt eingereichter Code lesende Zugriffe auf sensible Teile des Hauptspeichers, ist es ggf. auch hiermit möglich, sensible Informationen und Geheimnisse sowie Musterlösungen auszulesen.
- Falls eine Abgabe Zugriff auf das Netzwerk und damit auch auf andere Systeme besitzt, besteht die Gefahr, dass ein Angreifer sensible Informationen über diese erlangt. Letzteres kann über direkte Kommunikation mit den Komponenten erfolgen oder durch Mitschneiden des gesamten Netzwerkverkehrs. Auch das Ausspionieren fremder Abgaben und Ergebnisse wäre so möglich.

- Hat eingereichter Code Zugriff auf die Interprozesskommunikation fremder Komponenten, können auch hierüber sensible Informationen, Geheimnisse oder Lösungen mitgelesen werden.
- Durch das Ausnutzen von Sicherheitslücken im Betriebssystem oder in anderen Softwarekomponenten kann es einem Angreifer auch möglich werden, Informationen auszuspionieren.

3.4. Vorhandene Sicherheitsmaßnahmen

In Grappa selbst sind derzeit keine Sicherheitsmaßnahmen vorgesehen, welche den eingereichten Code vom Rest des Systems isolieren. Es existiert lediglich ein Konfigurationsparameter, um die maximale Ausführungsdauer für eine Bewertung festzulegen. Die Gesamtumgebung ist damit kaum geschützt. Stattdessen wird ein Backend-Plugin direkt in der JVM ausgeführt, welche auch Grappa als Ausführungsumgebung dient. Je nach verwendeten Grader und Implementierung des Backend-Plugins kann die Bewertung (und damit auch die Ausführung des eingereichten Codes) entweder in der identischen JVM oder in einer externen Umgebung erfolgen.
Aus diesen Gründen wird zur Zeit die graderspezifische Implementierung von Sicherheitsmaßnahmen verwendet. Mögliche Sicherheitsmaßnahmen und deren Grenzen sind hierbei sehr stark von der verwendeten Programmiersprache und Plattform abhängig. Eine Programmiersprache könnte bereits eine Rechteverwaltung und Verbotsstrategien nativ, eine zweite nur über Umwege mit zusätzlichen Tools und eine dritte gar nicht unterstützen. Dies macht sowohl das Realisieren einer umfassenden Absicherung der Ausführungsumgebung für bestimmte Grader problematisch, als auch eine einheitliche Verwaltung und Pflege aufwendig.

Von Graja bereitgestellte Sicherheitsmaßnahmen

Graja's Backend-Plugin verhindert, dass die Bewertung direkt in Grappa's JVM ausgeführt wird und erstellt stattdessen eine separate JVM zur Bewertung. Hierzu wird aus dem Backend-Plugin zunächst nur ein Starter-Modul benutzt, welches dafür sorgt, dass alle Dateien entpackt vorliegen. Anschließend wird ein neuer Prozess mit dem Graja-Hauptmodul zur Bewertung gestartet.
Unerlaubte Aktionen zur Laufzeit werden von Graja mit Hilfe der Java Sicherheitsarchitektur unterbunden. Ein Aufgabenautor kann die Erlaubnisanforderungen durch das Hinterlegen von Policies definieren.
Graja ist zudem in der Lage, die verwendeten Systemressourcen zu limitieren. Dies umfasst die maximale Ausführungszeit in Sekunden sowie den maximal nutzbaren Haupt- und Festplattenspeicher, ebenfalls definierbar durch den Aufgabenautor. (Siehe [Gar16])

Insgesamt bietet Graja also Möglichkeiten, um umfangreiche Sicherheitsmaßnahmen für Java-Code zu implementieren. Die Sicherheit der Gesamtumgebung ist in der Praxis jedoch von den teils vom Aufgabenautor tatsächlich konfigurierten Restriktionen abhängig.

Vom Praktomaten bereitgestellte Sicherheitsmaßnahmen

Da es sich beim Praktomaten um eine auf Python basierende Webanwendung handelt, welche vom Backend-Plugin über HTTP kontaktiert wird, geschieht auch hier die Bewertung in einer externen Umgebung und nicht direkt in Grappa's JVM.
Um während der Bewertung von Java-Code unerlaubte Aktionen zu verhindern, stützt sich der Praktomat ebenfalls auf die Java Sicherheitsarchitektur. Diese ist für Abgaben so konfiguriert, dass nur Dateien im aktuellen Verzeichnis gelesen und geschrieben werden dürfen und ein Netzwerkzugriff nicht möglich ist.
Um die Bewertung von Python-Code teilweise abzusichern, kann eine virtuelle Python-Umgebung zum Einsatz kommen, welche die Importmöglichkeiten von eingereichtem Code einschränkt, indem in dieser nur eine minimale Anzahl an Paketen installiert wird.
Unabhängig von der zu bewertenden Programmiersprache ist es zudem möglich, die Bewertung im Kontext eines Benutzers mit eingeschränkten Rechten auszuführen. Die konkreten Rechte des Benutzers müssen dann zuvor manuell auf Betriebssystemebene definiert werden.
Eine weitere programmiersprachenunabhängige Möglichkeit der Absicherung im Praktomaten ist die Nutzung von Docker-Containern. Diese werden erst in Kapitel 5.1 näher erläutert.
Eine Beschränkung der maximalen Ausführungszeit über ein Konfigurationsparameter des Praktomaten ist ebenfalls möglich. (Siehe [Tos18], [BFPS17])

Insgesamt bietet der Praktomat also eine Vielzahl von Sicherheitsmaßnahmen, welche teilweise sogar programmiersprachenunabhängig sind. Die Maßnahmen sind jedoch nur optional und müssen separat aktiviert und ggf. an Gegebenheiten des Systems angepasst werden.

4. Soll-Zustand

In Kapitel 3 wurde der derzeitige Aufbau einer Grappa-Umgebung beleuchtet sowie mögliche Angriffsszenarien auf Autobewerter vorgestellt. Weiterhin wurde herausgestellt, dass jeder Grader derzeit eigene, unterschiedliche Sicherheitsmaßnahmen implementieren muss, sofern dies mit der genutzten Programmiersprache und Plattform überhaupt möglich ist. An dieser Stelle ist eine von Grappa angebotene Standardlösung wünschenswert, um eine umfassende Absicherung zu gewährleisten, welche außerdem leicht mit neuen/vorhandenen Graderanbindungen kombinierbar ist.
Im Folgenden werden die notwendigen Anforderungen an eine von Grappa bereitgestellte, abgesicherte Ausführungsumgebung für beliebige Grader und Programmiersprachen definiert.
Die nachfolgenden Anforderungen wurden aus den in Kapitel 3.3 definierten Angriffsszenarien und unter Zuhilfenahme von ebenfalls in [For06], [TB10], [MB12], [Lin14], [GBSO17], [Gar13] beschriebenen Gegenmaßnahmen ermittelt.

4.1. Funktionale Anforderungen

Umfassende Isolation der Ausführungsumgebung vom Rest des Systems

(1) Da bereits Angriffe während der Initialisierung/Kompilierung einer Einreichung möglich sind, muss neben der Ausführung und Bewertung auch dieses innerhalb der abgesicherten Ausführungsumgebung geschehen.

(2) Pro zu bewertender Abgabe wird genau eine Instanz der Ausführungsumgebung erstellt und anschließend wieder gelöscht. Dies verhindert eine Interaktion zwischen Abgaben und dass ggf. Modifikationen persistent bleiben. Hierdurch ist der Verwaltungs-Overhead pro Einreichung zwar etwas größer, dafür jedoch die Integrität der Ausführungsumgebung dauerhaft gesichert.

(3) Eine Abgabe hat außerhalb der Ausführungsumgebung keinen direkten lesenden, schreibenden oder ausführenden Zugriff auf fremde Dateisysteme. Dies verhindert das Ausspionieren von Informationen und dass das System in einen inkonsistenten Zustand gebracht wird. Hierdurch ist es von Nöten, dass sich alle externen Abhängigkeiten einer Programmieraufgabe auch in der Ausführungsumgebung befinden bzw. dort zusätzlich deployt werden müssen, welches auf Grund der deutlich erhöhten Sicherheit jedoch vertretbar ist.

(4) Die zur Ausführung des Autobewerters benötigten (Konfigurations-) Dateien und Bewertungsergebnisse können von Abgaben nicht beschrieben werden. Dies verhindert, dass der Autobewerter in einen inkonsistenten Zustand gebracht wird.

(5) Die für eine Abgabe verfügbaren Systemaufrufe sind auf ein Minimum beschränkt. Dies erschwert das Ausnutzen von Sicherheitslücken, aber verhindert auch, dass Systemkonfigurationen, Berechtigungen oder Capabilities geändert werden. Programmieraufgaben, welche umfangreiche Administrationen des Systems zur Aufgabe haben, sind somit nicht möglich, da die Integrität des Systems eine höhere Priorität hat.

(6) Die Verfügbarkeit von Netzwerkverbindungen und Internet innerhalb einer Ausführungsumgebung ist standardmäßig deaktiviert, aber für einzelne Zielhosts und Ports durch Konfiguration der Programmieraufgabe aktivierbar. Die Möglichkeit, Netzwerkverkehr mitzuschneiden, ist niemals vorhanden. Dies verhindert das Ausspionieren von Informationen. Programmieraufgaben, welche das Aufzeichnen von Netzwerkverkehr zur Aufgabe haben, sind somit nicht möglich, da die Vertraulichkeit des Systems eine höhere Priorität hat.

(7) Eine Abgabe hat keine Möglichkeit mit Prozessen außerhalb der Ausführungsumgebung über Interprozesskommunikation zu kommunizieren, sofern nicht eine entsprechende Netzwerkverbindung konfiguriert wurde. Auch der Eingriff in die interne Interprozesskommunikation des Autobewerters oder der Ausführungsumgebung darf nicht möglich sein. Dies verhindert das Ausspionieren von Informationen und Eingriffe in fremde Prozesse.

(8) Es ist einer Abgabe nur möglich, Signale an eigene Prozesse zu senden. Dies verhindert das Ausspionieren von Informationen und Eingriffe in fremde Prozesse.

(9) Es ist zu verhindern, dass eine Abgabe Zugriff auf sensible Teile des Hauptspeichers hat. Dies verhindert das Ausspionieren von Informationen und Eingriffe in den Bewertungsvorgang.

Konfigurierbare Beschränkungen von Systemressourcen

(1) Eine Instanz der Ausführungsumgebung wird nach einer konfigurierbaren Zeitspanne automatisch gestoppt und gelöscht. Dies verhindert, dass eine Abgabe dauerhaft Systemressourcen und Rechenzeit beansprucht. Hierfür ist es jedoch erforderlich, dass bei rechenintensiven Programmieraufgaben auch eine ausreichende Rechenzeit konfiguriert wird. Weiterhin sollte die entsprechend konfigurierte Zeitspanne nicht zu groß sein, um das System pro Abgabe nicht zu lange zu blockieren.

(2) Der für eine Instanz der Ausführungsumgebung verfügbare Arbeitsspeicher ist konfigurierbar. Dies verhindert eine übermäßige Beanspruchung des Arbeitsspeichers. Hierfür ist es jedoch erforderlich, dass bei Programmieraufgaben, welche viel Arbeitsspeicher benötigen, auch eine ausreichend große Kapazität konfiguriert wird.

Die konfigurierte Kapazität sollte allerdings in einem sinnvollen Verhältnis zum insgesamt verfügbaren Arbeitsspeicher des Systems stehen.

(3) Der für eine Instanz der Ausführungsumgebung verfügbare Festplattenplatz ist konfigurierbar. Dies verhindert eine übermäßige Beanspruchung der Festplatte. Auch hier ist es erforderlich, dass bei Programmieraufgaben, welche viel Festplattenplatz benötigen, auch eine ausreichend große Kapazität konfiguriert wird. Die konfigurierte Kapazität sollte allerdings in einem sinnvollen Verhältnis zum gesamten Festplattenplatz des Systems stehen.

(4) Die für eine Instanz der Ausführungsumgebung mögliche Auslastung der Netzwerkschnittstelle ist konfigurierbar. Dies verhindert eine übermäßige Beanspruchung der Netzwerkschnittstelle. Hierbei muss die Auslastung so gewählt werden, dass das System nicht vollständig blockiert werden kann, der Bewertungsvorgang durch eine langsame Netzwerkverbindung aber auch nicht unnötig verlangsamt wird.

Steuerung der Ausführungsumgebung

(1) Instanzen der Ausführungsumgebung können von außen (aber nur innerhalb von Grappa) gestartet, gestoppt und gelöscht werden.

(2) Innerhalb einer Instanz der Ausführungsumgebung kann die Bewertung angestoßen werden. Resultierende Ergebnisse, aber auch Fehler können anschließend ausgelesen werden. Die maximale Größe der auszulesenden (Fehler-) Ausgabe ist konfigurierbar, um das System nicht zu sehr auszulasten. Hierbei ist es vertretbar, dass ein sehr langer Log ggf. nicht vollständig gelesen wird.

(3) Pro Grader ist die maximale Anzahl gleichzeitig existierender Instanzen der Ausführungsumgebung konfigurierbar. Dies dient zur Lastbalancierung.

Planbare automatische Update-Funktion der Ausführungsumgebung

Um die Ausführungsumgebung und zugehörige Komponenten ohne manuellen Aufwand aktuell halten zu können, geschieht dies planbar und automatisiert. Sicherheitspatches erhalten so schnellen Einzug in das System, um zumindest einen Schutz gegen bekannt gewordene Sicherheitslücken zu erhalten.

4.2. Nichtfunktionale Anforderungen

Graderunspezifische Lösung

Jeder mögliche Grader kann mit geringem Konfigurationsaufwand auf eine standardisierte Art und Weise in die isolierte Ausführungsumgebung eingebettet werden.

Geringer Aufwand beim Starten von Instanzen der Ausführungsumgebung

Das Bewerten einer Abgabe dauert derzeit 10 bis maximal 30 Sekunden. Das Starten einer Instanz der Ausführungsumgebung und auch das Abholen von Ergebnissen dürfen diese Zeiten nicht deutlich verlängern.

Nutzung von etablierten, zukunftssicheren Technologien

Die Umsetzung soll mit Hilfe von etablierten Technologien, welche möglichst lange unterstützt werden, erfolgen. Aufwendige Anpassungen an der Ausführungsumgebung sollen so bei Versionswechseln verhindert werden.

Flexibles Administrieren unterschiedlicher Ausführungsumgebungen

Die zu entwickelnde Lösung soll die nötige Flexibilität bieten, sodass die Auswahl besteht, Ausführungsumgebungen verschiedener Grader entweder gemeinsam oder pro Grader zu administrieren.

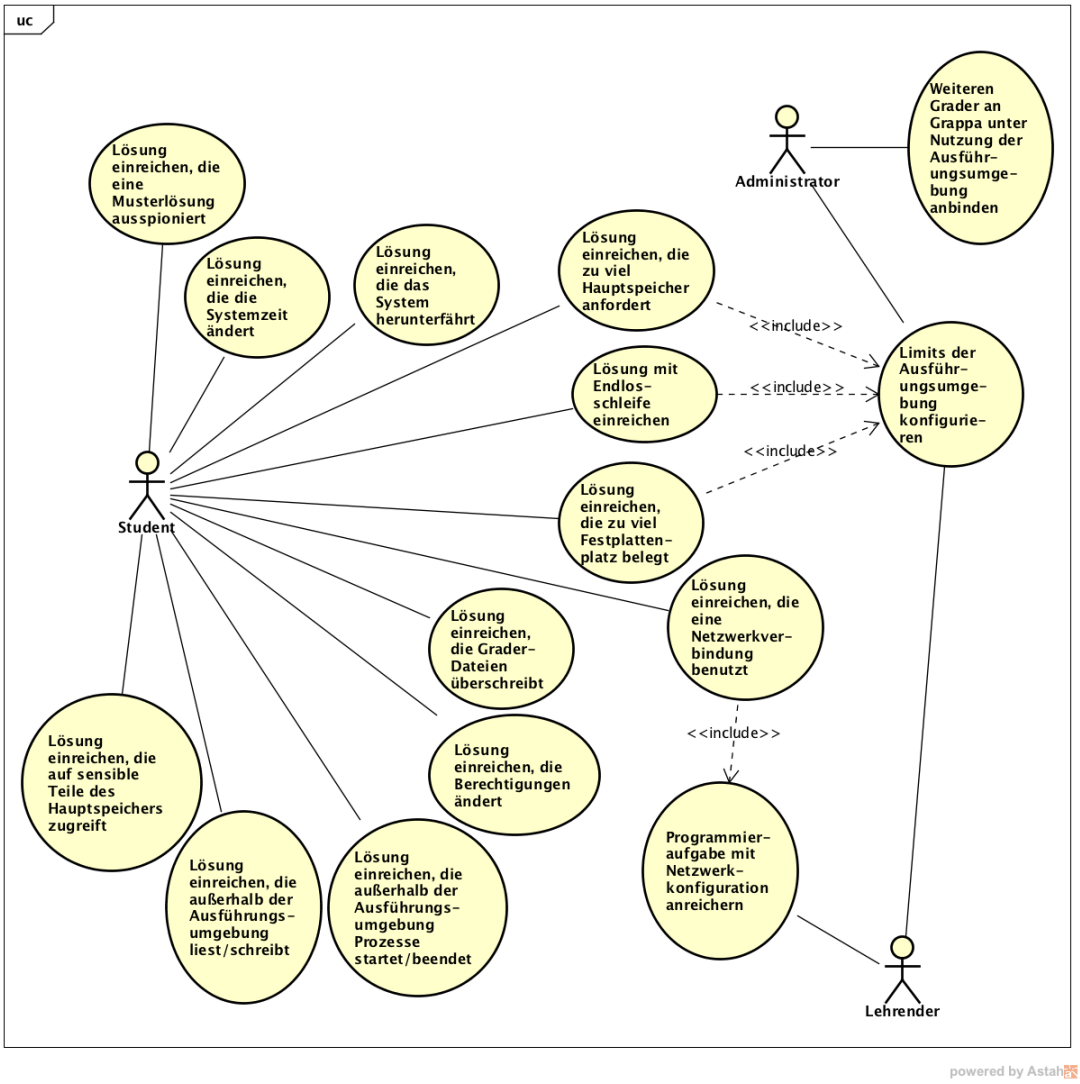


Abbildung 4.1.: Use-Case-Diagramm

4.3. Use-Cases

Die nun beschriebenen Use-Cases sind unabhängig vom verwendeten LMS und Grader formuliert und damit auch unabhängig von der verwendeten Programmiersprache und Plattform. Sie dienen jedoch als Vorlage für die in Kapitel 7 durchgeführten Tests für konkrete Grader. Die Abbildung 4.1 gibt zunächst eine Übersicht über alle Use-Cases.

- **Use-Case** „Weiteren Grader an Grappa unter Nutzung der Ausführungsumgebung anbinden“
  Akteur: Administrator

  Vorbedingung: Der Administrator hat bereits Zugriff auf eine funktionsfähige Grappa-Umgebung, an welche der Grader samt Ausführungsumgebung angebunden werden soll.
  Ereignisfluss:

  Standard:

  1. Der Administrator verschafft sich einen Überblick darüber, wie der Grader ohne Nutzung der Ausführungsumgebung installiert und angebunden wird.
  2. Der Administrator liest eine allgemeingültige Anleitung, welche keine tiefe Kenntnis über Implementierungsdetails der Ausführungsumgebung erfordert.
  3. Der Administrator installiert damit den Grader unter Zuhilfenahme von Konfigurationen auf die normale Art und Weise in die Ausführungsumgebung.

- **Use-Case** „Lösung mit Endlosschleife einreichen“
  Akteur: Student, Administrator, Lehrender

  Vorbedingung: Der Administrator/Lehrende hat ein Timeout für die Ausführungsumgebung hinterlegt. Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung und implementiert eine Endlosschleife in eine der von der Aufgabenstellung geforderten Methode.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit dem Hinweis, dass die Bewertung abgebrochen wurde.

  Alternativen:
  Zu 2: Anstatt einer Endlosschleife, baut der Student Kommandos ein, welche das System warten lassen (sleep).

- **Use-Case** „Lösung einreichen, die zu viel Hauptspeicher anfordert“
  Akteur: Student, Administrator/Lehrender

Vorbedingung: Der Administrator/Lehrende hat den maximalen Hauptspeicher für die Ausführungsumgebung hinterlegt. Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
Ereignisfluss:

Standard:

1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode immer mehr Hauptspeicher allokiert wird.
3. Der Student reicht die Lösung im LMS ein.
4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit dem Hinweis, dass die Bewertung abgebrochen wurde.

Alternativen:
Zu 2: Der Student präpariert eine Archivbombe als Lösung.

- **Use-Case** „Lösung einreichen, die zu viel Festplattenplatz belegt“
  Akteur: Student, Administrator, Lehrender

  Vorbedingung: Der Administrator/Lehrende hat den maximalen Speicherplatz für die Ausführungsumgebung hinterlegt. Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode immer mehr Daten auf die Festplatte geschrieben werden.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit dem Hinweis, dass die Bewertung abgebrochen wurde.

  Alternativen:
  Zu 2: Der Student präpariert eine Archivbombe als Lösung.

- **Use-Case** „Lösung einreichen, die eine Netzwerkverbindung benutzt“
  Akteur: Student, Lehrender

Vorbedingung: Der Lehrende hat die Programmieraufgabe nicht mit Netzwerkverbindungsinformationen angereichert. Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
Ereignisfluss:

Standard:

1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode versucht wird, auf die Google-Website (Port 80) zuzugreifen.
3. Der Student reicht die Lösung im LMS ein.
4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass die Google-Website nicht kontaktiert werden konnte.

Alternativen:
Zur Vorbedingung: Der Lehrende hat die Google-Website (Port 80) freigeschaltet.
Zu 4: Der Student erhält ein Bewertungsergebnis ohne Fehler.

- **Use-Case** „Lösung einreichen, die außerhalb des Ausführungsumgebung liest/schreibt“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode Dateien außerhalb der Ausführungsumgebung gelesen/geschrieben werden.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass an den entsprechenden Orten Dateien nicht gelesen und nicht geschrieben werden konnten.

- **Use-Case** „Lösung einreichen, die außerhalb der Ausführungsumgebung Prozesse startet/beendet“
  Akteur: Student

Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
Ereignisfluss:

Standard:

1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode ein Prozess außerhalb der Ausführungsumgebung gestartet/beendet wird.
3. Der Student reicht die Lösung im LMS ein.
4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass der Prozess nicht gestartet/beendet werden konnte.

- **Use-Case** „Lösung einreichen, die Grader-Dateien überschreibt“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode Dateien des Graders innerhalb des Arbeitsbereichs überschrieben werden.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass an den entsprechenden Orten Dateien nicht geschrieben werden konnten.

- **Use-Case** „Lösung einreichen, die Berechtigungen ändert“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.

2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode Berechtigungen auf Ordner und Dateien geändert werden.
3. Der Student reicht die Lösung im LMS ein.
4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass die Berechtigungen nicht geändert werden konnten.

- **Use-Case** „Lösung einreichen, die das System herunterfährt“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode ein Kommando zum Herunterfahren des Systems abgesetzt wird.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass das Kommando nicht ausgeführt werden konnte.

- **Use-Case** „Lösung einreichen, die die Systemzeit ändert“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode ein Kommando zum Ändern der Systemzeit abgesetzt wird.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass das Kommando nicht ausgeführt werden konnte.

- **Use-Case** „Lösung einreichen, die auf sensible Teile des Hauptspeichers zugreift“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode ein (programmiersprachenspezifischer) Dump des Hauptspeichers gemacht wird.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass die Aktion nicht ausgeführt werden konnte.

  Alternativen:
  Zu 2: Anstatt eines Dumps, schreibt der Student in den Hauptspeicher.

- **Use-Case** „Lösung einreichen, die eine Musterlösung ausspioniert“
  Akteur: Student

  Vorbedingung: Der Student ist bei einem LMS angemeldet und hat die Möglichkeit, eine Lösung zu einer Programmieraufgabe einzureichen. Weiterhin hat der Student Wissen darüber, wie und wo Musterlösungen des Graders gespeichert werden.
  Ereignisfluss:

  Standard:

  1. Der Student erstellt eine korrekte Lösung der Programmieraufgabe.
  2. Der Student bearbeitet seine Lösung, sodass in einer von der Lösung erwarteten Methode die Musterlösung gelesen und ausgegeben wird.
  3. Der Student reicht die Lösung im LMS ein.
  4. Der Student wartet kurz und erhält anschließend das Bewertungsergebnis mit der Exception, dass auf den Ablageort der Musterlösung nicht zugegriffen werden konnte.

5. Lösungsmöglichkeiten

Bevor konkrete Möglichkeiten und Technologien zur Umsetzung der Anforderungen betrachtet werden, gilt es zunächst zu klären, zu welchem Zeitpunkt im Bewertungsablauf und wo in der Gesamtarchitektur eine Absicherung der Umgebung geschehen sollte. [IAKS10], [Lin14] nennen hier einige Möglichkeiten, welche in realen Systemen im Autobewerterumfeld genutzt werden:

Serverseitiges, abgesichertes Ausführen einer Abgabe. Hier geschieht die Ausführung der Abgabe in der eigenen Serverumgebung. Eine Absicherung zur Laufzeit kann durch die Benutzung von erprobten Tools und Infrastruktur programmiersprachenunabhängig erfolgen. Dies wird auch als klassisches Sandboxing bezeichnet. Eine zuverlässige Isolation der Ausführungsumgebung ist dabei äußerst wichtig, um Angriffe auf die Serverinfrastruktur zu verhindern.

Statische Analyse vor der Ausführung einer Abgabe. Hier wird eine Abgabe zunächst einer statischen Analyse unterzogen. Nur wenn keine schädlichen Inhalte oder potenzielle Gefahren detektiert wurden, kommt es zur Ausführung und Bewertung der Abgabe. Eine solche statische Analyse ist nicht programmiersprachenunabhängig und aufwendig zu implementieren und zu pflegen.

Ausgelagerte Ausführung einer Abgabe. Hier geschieht die Ausführung der Abgabe nicht in der Serverumgebung, sondern entweder direkt auf dem System des Nutzers oder in von Dritten angebotenen Umgebungen. Somit sind Angriffe auf die eigene Serverinfrastruktur gar nicht erst möglich. Da jedoch keine umfangreiche Kontrolle über das ausführende System besteht, ist die Integrität und Verfügbarkeit des Bewertungsvorgangs schwierig zu gewährleisten.

Eine programmiersprachenunabhängige Lösung ist angestrebt und eine erhöhte Gefahr in den Bewertungsprozess einzugreifen, aus Sicht der definierten Anforderungen, nicht hinnehmbar. Aus diesem Grund werden im Folgenden nur Lösungsmöglichkeiten evaluiert, welche das klassische Sandboxing einer Abgabe zum Ziel haben.

Weiterhin beschränken sich die vorgestellten Technologien auf die Anwendung in Linux-Betriebssystemen, da diese im Autobewerterumfeld am weitesten verbreitet sind und auch Grappa derzeit produktiv in einer Ubuntu-Umgebung eingesetzt wird. Auf was für einem Betriebssystem ein Grader ausgeführt wird, ist hierbei nicht relevant, sofern die nötige Isolation zur Grappa-Umgebung gegeben und ein entsprechender Datenaustausch möglich ist.

5.1. Technologien zur Umsetzung des Sandboxings

5.1.1. Linux Features und Tools

Restriktion des Dateisystemzugriffs

In Unix erfolgt die Zugriffskontrolle auf das Dateisystem durch das Konzept von Benutzern und Gruppen. Bei einem Benutzer muss es sich nicht zwingend um eine echte Person handeln, welche das System nutzt. Es ist möglich, dass sogenannte System-Benutzer angelegt werden, um Prozesse/Daemons unter solch einem Benutzer auszuführen, damit das System vor unerwünschten Zugriffen geschützt ist.
Jede Datei und jedes Verzeichnis besitzt drei Zugriffstypen, für welche jeweils eigene Zugriffsrechte definiert werden können:

Owner: Zugriffsrechte des Benutzers, welcher die Datei besitzt. Es handelt sich meist um den Benutzer, der diese Datei erstellt hat.

Group: Zugriffsrechte der Gruppe, welche die Datei besitzt. Ein Benutzer kann mehreren Gruppen angehören.

Other: Zugriffsrechte für Benutzer, welche weder Besitzer der Datei noch Teil der Gruppe sind.

Die grundlegenden Zugriffsrechte können eine Kombination aus Lesen (read), Schreiben (write) oder Ausführen (execute) sein. Darüber hinaus gibt es erweiterte Zugriffsrechte, welche zur Ausführung eines Programms die Rechte des Owners (setuid) oder der Group (setgid) automatisch hinzufügen, sobald sie gesetzt sind.
Mit Hilfe von Benutzern und Zugriffsrechten lassen sich bereits alle Anforderungen umsetzen, welche eine Beschränkung von Dateizugriffen erfordern. Da in Linux auch die Prozess-Verwaltung (/proc) und einige Interprozesskommunikationsmechanismen (z.B. Named Pipes, Sockets) über das Dateisystem gesteuert werden, ist hier ebenfalls eine Restriktion möglich. Auch das Senden/Empfangen von Signalen kann in Unix nur innerhalb von Prozessen des gleichen Benutzers erfolgen und ist hiermit einschränkbar. (Siehe [GM84], [HH14], [MB12])

Unix bietet zudem den chroot Systemaufruf, mit dessen Hilfe das root-Verzeichnis aus Sicht eines Prozesses (und seiner Kinder) angepasst werden kann, um so eine eigene Datei-Umgebung (auch chroot jail genannt) für den Prozess zu schaffen. In ein chroot jail können beliebige Dateien und Verzeichnisse des echten Dateisystems eingebunden werden, um mögliche Zugriffe einzugrenzen. Es ist jedoch nötig, auch Laufzeitabhängigkeiten des Prozesses, wie z. B. Systembibliotheken, zu ermitteln und manuell einzubinden. Dies ist häufig mit viel Aufwand verbunden. (Siehe [Fri02], [HH14])

Restriktion von Prozessrechten

Neben dem Setzen von simplen Dateiberechtigungen unterstützt der Linux-Kernel das Setzen und Entziehen von Capabilities für ausführbare Dateien und Prozesse. Capabilities unterteilen die allumfassenden Berechtigungen des root-Benutzers in Untergruppen. Eine umfassende Liste und Beschreibung möglicher Capabilities kann der manpage entnommen werden. Capabillites ermöglichen eine feingranularere Zuweisung von privilegierten Zugriffsrechten und somit die weitere Einschränkung von unerlaubten Aktionen für eine Ausführungsumgebung. (Siehe [HM08])

Es existieren mehrere Tools, welche mit Hilfe des Systemaufrufs „ptrace“ in der Lage sind, Systemaufrufe eines Prozesses zu überwachen und zu kontrollieren. Ptrace-Tools agieren dabei als eine Art Debugger, welche bei jedem Aufruf einer Funktion des Kernels, den ausgeführten Prozess pausieren und den Aufruf inspizieren können. Ist der Systemaufruf als nicht zulässig eingestuft, wird der Prozess beendet, andernfalls kann die Ausführung fortgesetzt werden.
Beispielsweise wurde ein ptrace-Tool auch einige Zeit zur Absicherung des Autobewerters ASB genutzt, aufgrund von Sicherheitslücken jedoch wieder verworfen (siehe [GBSO17]). Ptrace-Tools sind weiterhin nicht in der Lage, Prozesse mit mehreren Threads zu überwachen. Dies stellt für Programmiersprachen, dessen Laufzeitumgebungen bereits mehrere Threads benötigen (wie z.B. Java), ein Problem dar. (Siehe [KMPP07], [MB12])

Des Weiteren existiert das Modul „seccomp“ im Linux-Kernel. Hierbei handelt es sich um ein Sicherheitsmodul, mit welchem das Einschränken erlaubter Systemaufrufe durch definierbare Filter möglich ist. Ein Prozess oder Thread muss hierfür lediglich einen Systemaufruf tätigen, um in einen durch das Sicherheitsmodul abgesicherten Zustand zu gelangen. Somit wird eine umfangreiche Einschränkung von unerlaubten Aktionen für eine Ausführungsumgebung möglich. (Siehe [Dre12])

Mandatory Access Control (MAC)

Bei Mandatory Access Control (MAC) handelt sich um eine zusätzliche, mächtigere Kontrollschicht neben der Standardrechteverwaltung von Unix. Inhalte eines Systems werden dabei als Subjekte und Objekte betrachtet, welche jeweils Sicherheitsattribute besitzen. Sobald ein Subjekt auf ein Objekt Zugriff erhalten will, werden die Sicherheitsattribute vom Kernel ausgelesen und anhand einer Policy entschieden, ob der Zugriff erfolgen darf. Bei Subjekten kann es sich beispielsweise um Prozesse/Threads handeln und bei Objekten z. B. um Dateien, Netzwerkports oder gemeinsam genutzte Speicherregionen. Auch eine Zuweisung erlaubter Capabilities ist möglich. Prominente Sicherheitsmodule, welche MAC ermöglichen, sind das bereits im Linux-Kernel integrierte SELinux oder AppArmor.
MAC ermöglicht eine äußerst umfangreiche und flexible Einschränkung eines Systems

(inklusive des root-Benutzers). MAC ist jedoch komplex und die Konfiguration deshalb oft mit hohem Aufwand verbunden. (Siehe [Lin06], [Nak07])

Limitierung von Ressourcen und Netzwerkeinschränkungen

Eine Beschränkung von CPU, RAM und I/O-Operationen ist in Linux mit cgroups möglich. Diese werden in Kapitel 5.1.3 näher erläutert.

Unix ermöglicht eine Begrenzung von Festplattenspeicher durch Disk Quota. Mit diesem Mechanismus ist es möglich, für Benutzer oder Gruppen ein maximales Speicherplatzlimit für Speichermedien festzusetzen. (Siehe [TB00])
Weiterhin ist es möglich sogenannte loop devices einzusetzen, um Speicherplatz zu limitieren. Hierbei wird eine normale Datei als (virtuelles) blockorientiertes Gerät in ein Dateisystem eingebunden. Das loop device lässt sich anschließend wie eine echte Festplattenpartition verwalten und benutzen. (Siehe [Wal07])

Der Linux Kernel bietet mit Netfilter ein Framework, welches eine umfangreiche Manipulation des eigenen Netzwerkverkehrs auf tiefer Ebene ermöglicht. Das Kernelmodul „iptables“ nutzt dieses Framework, um eine regelbasierte Firewall auf Kernel-Ebene zu realisieren. Mit Hilfe von entsprechend definierten Regeln lassen sich so erlaubte und verbotene Netzwerkverbindungen definieren. (Siehe [KWM+04])

Traffic Shaping ist die Fähigkeit, Netzwerkverbindungen verschiedene Prioritäten zuweisen zu können. Technisch ist dies mit einer Verzögerung von Netzwerkpaketen durch Warteschlagen möglich. Traffic-Shaping kann so eine Begrenzung der Datenrate und damit eine geringere Auslastung einer Netzwerkschnittstelle bewirken. In Linux kann das Kommando „tc“ zur Definition von Regeln für ein Traffic-Shaping benutzt werden. (Siehe [Sta01])

5.1.2. Virtuelle Maschinen

Virtuelle Maschinen können als eigenständige Systeme mit einem eigenen Betriebsystem und Kernel aufgefasst werden. Eigene Restriktionen innerhalb virtueller Maschinen sind so zusätzlich möglich.
Virtuelle Maschinen stehen unter Kontrolle eines Virtual Machine Monitors (VMM). Der VMM oder auch Hypervisor genannt, stellt für sie die gesamte Umgebung in Form von virtueller Hardware zur Verfügung und fügt somit gleichzeitig eine Isolation auf tiefer Ebene hinzu. Es existieren zwei verschiedene Typen von Hypervisoren (siehe Abb. 5.1):

Typ 1: Hier wird der VMM direkt auf der Hardware ausgeführt, sodass die Ausführung einer virtuellen Maschine besonders effizient ist und wenig Angriffsfläche geboten wird.

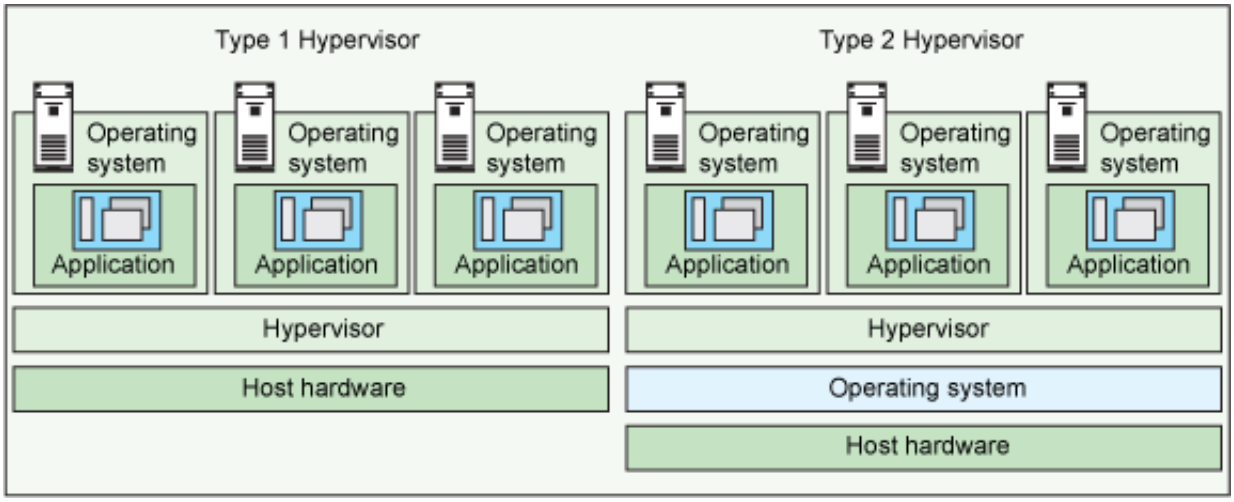


Abbildung 5.1.: Hypervisor Typen (Quelle: [Tho11])

Typ 2: Hier wird der VMM als Anwendung auf einem Host-Betriebssystem ausgeführt. Dies macht die Ausführung einer virtuellen Maschine etwas weniger effizient und bietet etwas mehr Angriffsfläche. Das Aufsetzen von virtuellen System innerhalb eines Host-Betriebssystems ist jedoch mit wenig Aufwand zu erreichen und an weniger Hardwarelimitationen gebunden.

Es existieren diverse Hersteller, welche Virtualisierungssoftware anbieten. Beispiele sind hier

QEMU/KVM QEMU ist ein Emulator für CPUs und KVM ein Kernel-Modul. Gemeinsam ermöglichen sie die Virtualisierung über den Linux-Kernel.

VMWare Hierbei handelt es sich um einen Hersteller, welcher verschiedene kommerzielle Virtualisierungsprodukte anbietet.

VirtualBox Dies ist eine frei verfügbare Virtualisierungssoftware.

Eine umfangreich kontrollierbare Zuteilung und Limitierung von Ressourcen kann meist über die verwendete Virtualisierungssoftware erfolgen.
Da ein Hypervisor Funktionalitäten nachbilden muss, welche unter normalen Umständen von physikalischer Hardware erledigt werden, ist das Starten und Betreiben von virtuellen Maschinen ressourcenintensiv und ihr Design damit schwergewichtig. Des Weiteren muss die Konfiguration und Aktualisierung von virtuellen Maschinen manuell erfolgen, sofern keine zusätzlichen Tools zum Einsatz kommen. Durch die Verwendung eines Hypervisors ist eine virtuelle Maschine effektiv abgeschottet. Schadcode müsste zunächst aus der virtuellen Maschine ausbrechen und über den Hypervisor das unterliegende System angreifen, um eine große Bedrohung darzustellen. (Siehe [SN05], [RG05], [Lin14])

Vagrant

Bei Vagrant handelt es sich um eine Open-Source-Software des Unternehmens HashiCorp mit der Funktion, portable virtuelle Umgebungen aufzusetzen und zu verwalten. Vagrant setzt hierzu auf eine vorhandene Virtualisierungssoftware auf. Es ermöglicht eine automatisierte Steuerung dieser durch das Hinzufügen benutzerfreundlicher, standardisierter Mechanismen. Vagrant wurde 2010 erstmalig in einer stabilen Version veröffentlicht und ist über ein CLI steuerbar.
Die zentralen Elemente bei der Benutzung von Vagrant sind:

Boxes Dies sind vorkonfigurierte, paketierte Umgebungen. Im Normalfall handelt es sich um eine virtuelle Maschine, in welche ein Betriebssystem, Bibliotheken und Tools vorinstalliert sind. Eine Box wird entweder selbst erstellt oder aus einem Katalog in der Cloud bezogen. Anschließend können Boxes in einem lokalen Repository, welches Teil einer Vagrant-Installation ist, importiert und verwaltet werden. Beispielsweise stellen die Entwickler von Ubuntu eine aktuelle Vagrant-Box ihres Betriebssystems über den standardmäßigen Cloud-Katalog von Vagrant zur Verfügung.

Vagrantfiles Bei einem Vagrantfile handelt es sich um eine Konfigurationsdatei, in welcher eine zu verwendende Basis-Box sowie Konfigurations- und Provisioning-Details hinterlegt sind. Wird im Verzeichnis eines Vagrantfiles der Befehl „vagrant up“ ausgeführt, so wird die konfigurierte Umgebung erstellt und gestartet.

Vagrant erleichtert die Verwaltung und Handhabung virtueller Umgebungen und macht diese portabel. Benutzer müssen sich so nicht mehr mit herstellerspezifischen Details auseinandersetzen. Der Ressourcenverbrauch und Overhead, welcher beim Starten und Betreiben einer virtuellen Maschine anfällt sowie Sicherheitsaspekte, ändern sich durch die Nutzung von Vagrant nicht. (Siehe [Has13], [Has18])

5.1.3. Container

Bei Containern handelt es sich vereinfacht ausgedrückt um eigenständige isolierte Partitionen eines Betriebssystems. Um eine solche Partitionierung bzw. einen Container zu ermöglichen, werden in Linux sogenannte Kernel Namespaces und Control Groups verwendet. Container teilen sich also den Kernel mit einem Host-Betriebssystem.
Kernel Namespaces erlauben das Ausführen von Prozessen (und deren Kinder) isoliert innerhalb von Namensräumen. Namensräume kapseln jeweils eine Menge von Betriebssystemressourcen gleicher Art voneinander. Aus Sicht eines Prozesses ist es nicht erkennbar, dass dieser innerhalb eines Namensraums ausgeführt wird. Linux bietet die folgenden Namespaces, um Containern die Isolation verschiedener Komponenten zu gewährleisten:

IPC ermöglicht die Nutzung von voneinander unabhängiger, isolierter Interprozess-Kommunikations-Mechanismen.

MNT ermöglicht eine eingeschränkte Sicht auf das reale Dateisystem.

NET ermöglicht die Nutzung eines eigenen vom Host-Betriebssystem unabhängigen Netzwerkstacks.

PID ermöglicht die Nutzungs eines eigenen Prozess-Subsystem mit eigenen Prozess-IDs und eigener PID 1 (Urvater Prozess).

USER ermöglicht eigene vom Host-Betriebssystem unabhängige Benutzer und Gruppen sowie eigene interne Capabilities und interne privilegierte Benutzer.

UTS ermöglicht einen eigenen Host- und Domänennamen.

(Siehe [LP17], [MB12])

Control Groups ermöglichen es, einzelne oder miteinander korrespondierende Prozesse zu Gruppen (cgroups) zusammenzufügen. Daraufhin kann mit Hilfe eines sogenannten Control Group Subsystem eine Begrenzung, Zuordnung oder Messung von Ressourcen für eine cgroup erfolgen. Einige wichtige Subsysteme, um Denial of Service Angriffe aus einem Container heraus zu verhindern, sind:

cpuset Hiermit können einer Gruppe bestimmte Prozessorkerne und Memory Nodes zugeordnet werden.

cpu Hiermit können Scheduler Einstellungen für die Gruppe festgelegt werden.

blkio Hiermit können I/O-Operationen für blockorientierte Geräte für die Gruppe eingeschränkt werden.

memory Hiermit kann der nutzbare RAM für die Gruppe limitiert werden.

(Siehe [LP17], [MB12])

Mit Hilfe von Kernel Namespaces und cgroups lassen sich bereits rudimentäre Container und damit über den Kernel umfangreich isolierte Ausführungsumgebungen implementieren. Letzteres ist beispielsweise auch im Rahmen von [MB12] erfolgt. Im Gegensatz zu einer virtuellen Maschine benötigt ein Container keine virtuelle Hardware und kein eigenes Betriebssystem, sodass ein wesentlich schlankeres Design vorliegt. Auch das Starten und Betreiben von Containern ist somit deutlich ressourcenschonender. Da sich Container jedoch den Kernel mit einem Host teilen und zahlreiche, nicht komplexe Angriffsszenarien existieren/existierten, um aus Namespaces auszubrechen (siehe [Ker13]), ist eine rudimentäre Containerlösung weniger sicher als eine virtuelle Maschine. Abbildung 5.2 visualisiert die Isolation von Containern über den Kernel.

Da das manuelle Aufsetzen solcher Umgebungen mit Low-Level-Tools sehr aufwendig ist und tiefes Fachwissen erforderlich ist, existieren diverse Spezifikationen und Plattformen, welche Container portabel und das Aufsetzen und Betreiben dieser benutzerfreundlicher

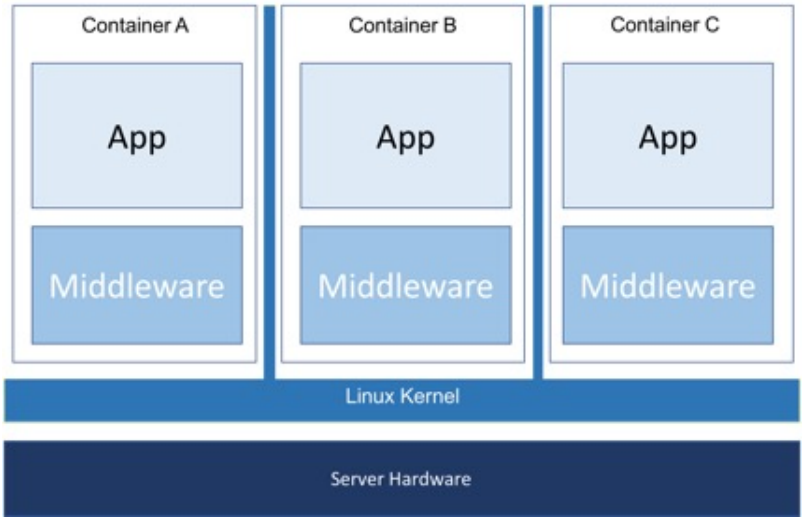


Abbildung 5.2.: Container Isolation (Quelle: [Fou18])

machen. Zudem bieten solche Plattformen weitere Funktionen und zusätzliche Sicherheitsmaßnahmen. Ein Container verhält sich damit wie ein eigenständiges System, innerhalb dessen wiederum eigene Restriktionen möglich sind.
Vor allem in den Bereichen kontinuierliche Integration, DevOps, Automation und horizontale Skalierbarkeit erhielten Container-Technologien in den letzten Jahren viel Aufmerksamkeit. Im Folgenden werden die populärsten und aktuellsten Plattformen mit Hauptaugenmerk auf Sicherheitsaspekte vorgestellt.

Docker

Bei Docker handelt es sich um die derzeit weitverbreiteste und bekannteste Container-Plattform. Sie ist seit 2014 in einer stabilen Version verfügbar und wird vom Unternehmen Docker, Inc. entwickelt. Docker kann sowohl über ein CLI, als auch über eine REST-API gesteuert werden. Die bei der Benutzung von Docker wesentlichsten Konzepte sind:

Images Ein Image enthält Userland-Komponenten eines Betriebssystems (keine Kernel-Komponenten) und ggf. zusätzlich installierte Bibliotheken und Tools. Images bilden die Grundlage einer Container-Instanz und sind read-only. Sie können entweder über eine externe Datenbank (sogenannte Registries) bezogen oder selbst erstellt werden. Images sind schichtenweise aufgebaut, sodass ein vorhandenes Image durch das Hinzufügen einer weiteren Schicht um weitere Inhalte erweitert werden kann. Images enthalten zudem einen Einstiegspunkt, welcher bei der Ausführung einer Container-Instanz zum Tragen kommt.

Dockerfiles Dockerfiles sind Dateien, mit dessen Hilfe die Basis und Inhalte eines Images genauestens spezifiziert werden können. Aus einem Dockerfile kann Docker ein

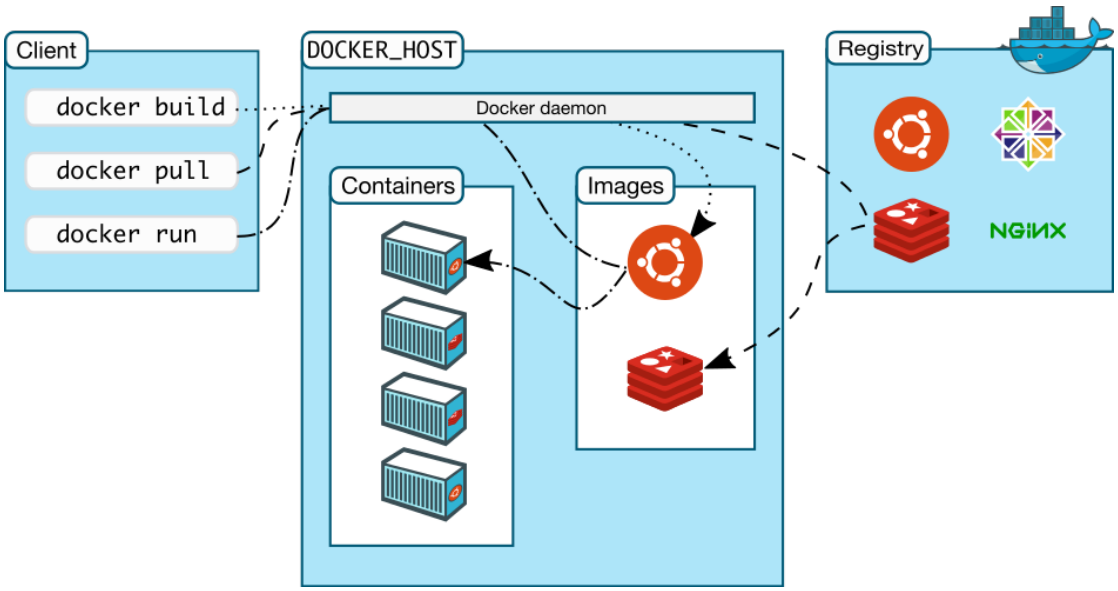


Abbildung 5.3.: Docker Architektur (Quelle: [Inc18])

Image erzeugen.

Container Container sind ausführbare Instanzen von Images mit einem eigenen Arbeitsbereich. Hierzu wird eine eigene von der Instanz beschreibbare Schicht auf das Image gelegt.

Docker ist in einer Client-Server-Architektur organisiert. Der Docker Daemon wird auf einem Host-System, auch Container-Host genannt, ausgeführt. Er nimmt als zentrale Verwaltungsinstanz API-Anfragen von Clients entgegen, um Images und Container-Instanzen erstellen und verwalten zu können (siehe Abb. 5.3). Soll es zu der Ausführung eines Containers kommen, geschieht dies seit Version 1.11 durch die Delegation an die Container-Laufzeitumgebung und -Bibliothek „runC“ (siehe Abb. 5.6). Bei runC handelt es sich um eine Implementierung der OCI (Open Container Initiative)-Spezifikationen zur Vereinheitlichung von Container-Formaten und -Laufzeitumgebungen (siehe [Ini17]). Um Container neben der Nutzung von Namespaces und cgroups weiter abzusichern, bietet Docker in der aktuellen Version:

- Eine Einschränkung der Capabilities, mit denen eine Container-Instanz ausgeführt wird. Eine per Default definierte Einschränkung ist bereits aktiv.
- Eine Einschränkung der in einer Container-Instanz verfügbaren Systemaufrufe durch seccomp. Ein vordefinierter Filter ist per Default bereits aktiv.
- Eine Absicherung von Container-Instanzen mit Mandatory Access Control (MAC) durch AppArmor. Ein vordefiniertes AppArmor Profil ist bereits per Default aktiv.
- Eine Sicherung der Integrität von Images durch die eigens entwickelten Mechanismen Content Trust und Notary.

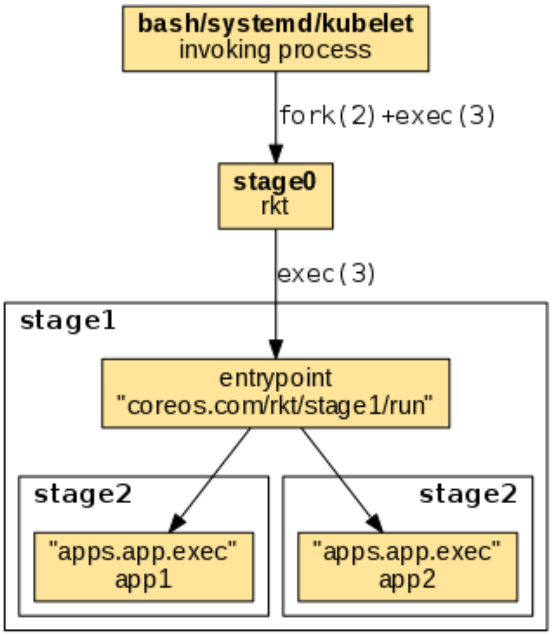


Abbildung 5.4.: Rkt Ausführungskette (Quelle: [RH18])

Einige Ressourcenlimits können zudem pro Container bequem definiert werden, ohne direkt mit anderen Tools interagieren zu müssen. (Siehe [LP17], [Inc18])

rkt

Bei rkt handelt es sich um eine weniger populäre Alternative zu Docker, welche seit 2016 in einer stabilen Version verfügbar ist. Das Konzept und der Funktionsumfang von rkt ist im Vergleich zu aktuellen Docker-Versionen nahezu identisch. Rkt kann sowohl mit Docker-Containern arbeiten, als auch mit dem eigenen (nicht OCI konformen und nicht mehr weiterentwickelten) Appc-Container-Format. Die Steuerung von rkt erfolgt über CLI-Tools oder über einen API-Service in gRPC. Rkt ist eine Eigenentwicklung von CoreOS für ein gleichnamiges Container-Host-Betriebssystem, welches das Clustering von Container-Hosts vereinfachen soll.
Der wesentliche Unterschied zu Docker ist die Architektur, da es keinen zentralen Daemon zur Verwaltung gibt. Das Erstellen von Images erfolgt mit dem Kommandozeilentool „actool“ und das Administrieren/Ausführen von Images und Container-Instanzen mit dem Kommandozeilentool „rkt“.
Rkt galt lange Zeit als sicherere Alternative zu Docker und wird derzeit auch weiterhin als solche vermarktet, da Docker in früheren Versionen keine zusätzlichen Mechanismen zur Absicherung anbot. Die aktuellen Sicherheitsvorteile beschränken sich jedoch auf den Verzicht des zentralen, mit privilegierten Rechten ausgeführten Daemons (siehe Abb. 5.4) und einer höheren Modularität der Sicherheitsarchitektur. Um Container

neben der Nutzung von Namespaces und cgroups weiter abzusichern bietet rkt nahezu gleiche Maßnahmen:

- Eine Einschränkung der Capabilities, mit denen eine Container-Instanz ausgeführt wird. Eine per Default definierte Einschränkung ist bereits aktiv.
- Eine Einschränkung der in einer Container-Instanz verfügbaren Systemaufrufe durch seccomp. Ein vordefinierter Filter ist per Default bereits aktiv.
- Eine optionale Absicherung von Container-Instanzen mit Mandatory Access Control (MAC) durch SELinux.
- Eine Sicherung der Integrität von Images durch die Möglichkeit diese mit GPG zu signieren.

Einige Ressourcenlimits können auch hier pro Container bequem definiert werden, ohne direkt mit anderen Tools interagieren zu müssen. (Siehe [RH18], [LP17])

Kata Containers

Kata Containers ist ein Open-Source-Projekt mit der Zielsetzung Container und leichtgewichtige virtuellen Maschinen miteinander zu kombinieren. Damit soll eine Technologie bereitgestellt werden, welche die Performanz und Flexibilität von Containern und zusätzlich eine erhöhte Sicherheit durch die Isolation mit einem Hypervisor bietet. Das Projekt ist ein Zusammenschluss der Technologien „Intel Clear Containers“ und „Hyper runV“, welche vor Kata Containers jeweils eine ähnliche Zielsetzung hatten. Kata Containers wurde erstmals in Version 1.0 im Mai 2018 veröffentlicht und kann durch eine OCI konforme Implementierung in Docker als Container-Laufzeitumgebung gesetzt und benutzt werden (siehe Abb. 5.6).
Sobald Kata Containers genutzt wird, wird für jede Container-Instanz eine eigene virtuelle Maschine mit einem minimalen Linux-Kernel gestartet. Als Hypervisor kommt hierbei die vom Linux-Kernel unterstützen Lösungen QEMU/KVM zum Einsatz. Innerhalb des minimalen Linux-Kernel wird dann die eigentliche Container-Instanz gestartet (siehe Abb. 5.5).
Kata Containers bietet durch Virtualisierung eine Absicherung auf tieferer Ebene. Die Technologie ist dabei im Gegensatz zu virtuellen Maschinen deutlich ressourcenschonender und kann bequem mit Docker gesteuert werden. Da sich das Projekt derzeit jedoch noch in einem relativ frühen Stadium befindet, ist eine umfangreiche Dokumentation noch nicht verfügbar. Auch einige Operationen, welche Docker auf normalen Containern ausführen kann, werden von Kata Containers derzeit noch nicht unterstützt. Diese umfassen beispielsweise das Festlegen von Ressourcenlimits oder die Interaktion mit dem Dateisystem eines Containers von außen. (Siehe [Fou18])

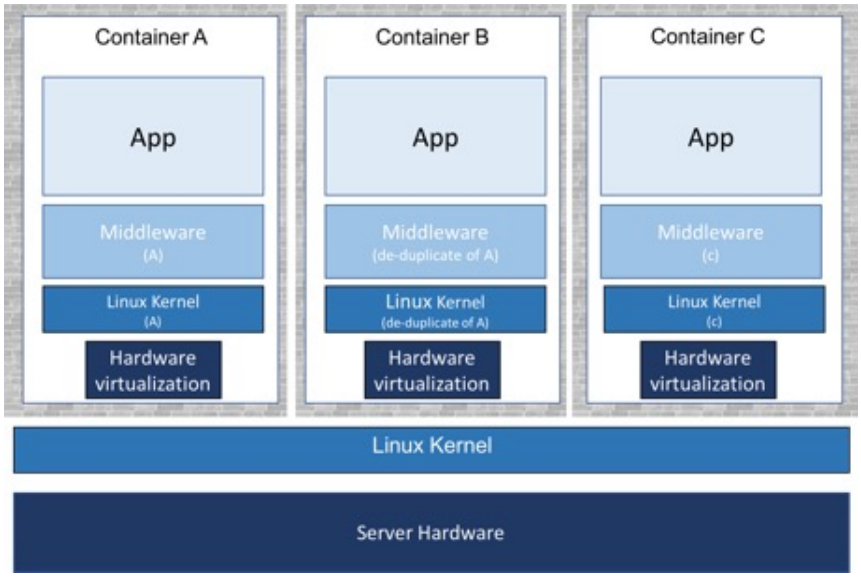


Abbildung 5.5.: Kata Containers Isolation (Quelle: [Fou18])

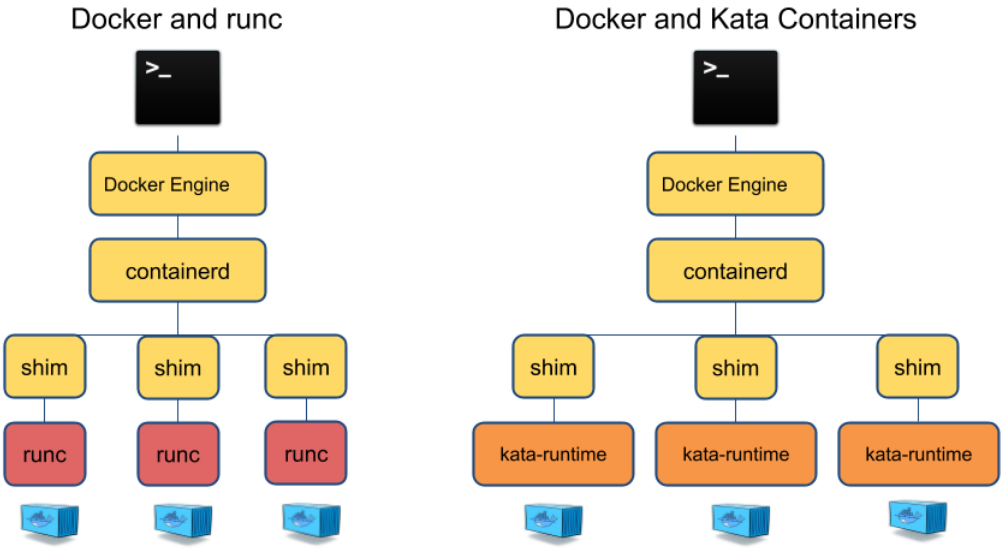


Abbildung 5.6.: Docker Container-Laufzeitumgebungen (Quelle: [Fou18])

	Linux Features und Tools	**Virtuelle Maschinen**	**Container**
Einschränkungs-möglichkeiten	Vom Tool abhängig	Komplett eigenständiges System, Ressourcenlimits	Dateisystem, Prozesse, Netzwerk, IPC, Benutzer, Capabilities, Systemaufrufe, Ressourcenlimits
Komplexität/ Konfigurations-aufwand	Hoch (mehrere Tools notwendig)	Mittel (Vagrant: Gering)	Gering (Manuell mit Namespaces + cgroups: Sehr hoch)
Isolationsebene	Kernel	Hardware	Kernel (Kata Containers: Hardware)
Zusätzlicher Ressourcen-verbrauch	Minimal	Hoch	Gering
Etabliert/ zukunftssicher	Ja	Ja	Nur Docker

Tabelle 5.1.: Vergleich der Technologien zum Sandboxing

Technologieauswahl

In Tabelle 5.1 ist eine Übersicht über die betrachteten Technologien und deren Eigenschaften abgebildet.
Ein Sandboxing und Ressourcenlimitierung mit Linux Features und Tools (inklusive Namespaces und cgroups) ist nur durch eine Kombination dieser möglich, da keine universelle Lösung existiert. Zudem handelt es sich teilweise um Low-Level-Mechanismen für weitgefasste Anwendungsgebiete, sodass die Entwicklung und Pflege einer Ausführungsumgebung mit großem Aufwand verbunden und viel Fachwissen über Interna erforderlich ist. Kleine Fehler bei der Implementierung führen so ggf. zu einer hohen Beeinträchtigung der Sicherheit. Daher sollte besser zu einer vorgefertigten Lösung gegriffen werden, auch wenn der Ressourcen-Overhead mit einer Eigenentwicklung am kleinsten wäre.
Das Nutzen von virtuellen Maschinen als Ausführungsumgebung bietet die höchstmögliche Sicherheit durch Hardwareabstraktion. Zudem wird durch Tools wie Vagrant die Wahl des spezifischen Virtualisierungsproduktes weniger relevant und die Handhabung von virtuellen Maschinen komfortabel. Der Ressourcen-Overhead ist jedoch zu hoch, sodass nur wenige zeitgleiche Bewertungen ausgeführt werden können und Bewertungen durch den Initialisierungsprozess deutlich längere Zeiten benötigen. Das alleinige Nutzen von virtuellen Maschinen als Ausführungsumgebung pro Abgabe ist im Bezug auf Performanz-Anforderungen daher nicht praktikabel.
Vorgefertigte Container-Lösungen bieten umfangreiche Isolationsmaßnahmen über den Kernel, eine Integration zusätzlicher Sicherheitsmaßnahmen, eine komfortable Handhabung und geringen Ressourcen-Overhead. Vor allem Kata Containers wäre durch die zusätzliche Hardwareabstraktion daher die optimale Technologie, jedoch bzgl. Dokumentation und Funktionsumfang in einem zu frühen Stadium. Docker ist aufgrund seiner

OCI konformen Komponenten und deutlich größeren Marktanteil zukunftssicherer und etablierter als rkt, auch wenn rkt eine etwas sicherere Architektur bietet. Daher sollte Docker als Technologie zur Umsetzung dienen.

Um Docker einen aktuellen Kernel als Grundlage für Container-Instanzen zur Verfügung zu stellen und auch den mit privilegierten Rechten ausgestatteten Daemon vom restlichen System zu isolieren, bietet es sich an, zusätzlich eine virtuelle Maschine (Typ 2) als Container-Host zu verwenden. Somit ist eine starke Isolation über Hardwareabstraktion zur Serverumgebung und eine etwas schwächerer Isolation über den Kernel zwischen den Container-Instanzen gegeben. Das verwendete Linux-Betriebssystem ist hierbei nicht relevant, solange es von der aktuellen Docker-Version unterstützt wird. Es bietet sich jedoch Ubuntu an, da es derzeit produktiv auf dem Grappa-Server zum Einsatz kommt.

Zudem können weitere Tools innerhalb von Container genutzt werden, um zusätzliche Funktionalitäten zu ermöglichen, welche Docker selbst nicht anbietet. Sinnvolle zusätzliche Funktionalitäten sind bzgl. der Anforderungen die Unterstützung von Firewall-Regeln und Traffic Shaping.

5.2. Steuerung der Ausführungsumgebungen

Steuerung von Docker

Der Docker Daemon kann entweder über das Kommandozeilentool „docker“ oder über eine REST-API gesteuert werden. Beide Möglichkeiten bieten eine Kontrolle über Images und Container-Instanzen im gleichen Umfang. Die Nutzung der REST-API ist hierbei vorzuziehen, da es sich um einen standardisierten Kommunikationsmechanismus zwischen Applikationen handelt.
Docker bietet bereits offizielle SDKs in den Programmiersprachen Python und Go an, welche die REST-API Aufrufe in einer eigenen Bibliothek kapseln und somit eine bequeme Steuerung aus der Programmiersprache heraus ermöglichen. Zudem listet Docker weitere inoffizielle Bibliotheken für andere Programmiersprachen. Für Java werden hier die Bibliotheken „docker java“ (siehe [dj18]) und „Docker Client“ (siehe [Spo18]) genannt, welche beide unter der Apache Lizenz 2.0 verfügbar sind. Während docker-java bisher nur eine Teilmenge der Docker API zur Verfügung stellt, bietet Docker Client eine nahezu komplette Implementierung der API. Docker Client wird zudem im Open-Source-Respository des Unternehmens Spotify verwaltet und befindet sich laut eigener Angaben dort im produktiven Einsatz.
Da es sich bei Docker Client um eine gut getestete Java-Bibliothek handelt, eignet sich diese optimal zur Steuerung von Docker aus Grappa heraus und macht eine direkte Nutzung der Docker REST-API über HTTP nicht notwendig. (Siehe [Inc18])

Steuerung des Container-Hosts

Um den Container-Host zu steuern, kann entweder ein vom Virtualisierungshersteller bereitgestellter Mechanismus oder das bereits vorgestellte Vagrant zum Einsatz kommen. Um eine höhere Flexibilität und eine Austauschbarkeit des Virtualisierungsproduktes zu gewährleisten, ist das herstellerübergreifende Management-Tool die bessere Wahl.
Vagrant besitzt keine API-Schnittstelle zur Steuerung, deshalb muss direkt das Kommandozeilentool genutzt werden. Aufgrund der wenig komplexen CLI-Kommandos, stellt dies jedoch keine zusätzlichen Herausforderungen dar. (Siehe [Has18])

Übermitteln einer Abgabe und Abholen des Bewertungsergebnisses

Damit eine Abgabe innerhalb der Ausführungsumgebung bzw. einer Container-Instanz bewertet werden kann, ist ein Kommunikationsweg zum Übermitteln einer Abgabe und dem Abholen des Bewertungsergebnisses erforderlich.
Eine naheliegende Lösungsmöglichkeit wäre hier das Einrichten einer Interprozesskommunikation zwischen einer Container-Instanz und Grappa. Da eine Interprozesskommunikation nach außen, wie in Kapitel 3.3 beschrieben, einen Angriffspunkt darstellt, sind solche Mechanismen möglichst zu vermeiden.
Mit Hilfe sogenannter Bind Mounts oder Volumes ist es mit Docker möglich, Verzeichnisse und Dateien des Host-Betriebssystems innerhalb Container bereitzustellen. Über ein gemeinsames Verzeichnis könnte so der Austausch von Abgabe und Bewertungsergebnis auf Dateisystemebene erfolgen. Dieser Lösungsansatz würde die Isolation der Ausführungsumgebung leicht aufbrechen, da so auch zur Laufzeit potenzieller Zugriff auf ein Verzeichnis außerhalb der eigenen Umgebung bestünde.
Docker bietet zudem die Möglichkeit Inhalte direkt vom oder zum Dateisystem einer Container-Instanz zu kopieren. Dabei ist es nicht relevant, ob sich eine Container-Instanz gerade in der Ausführung befindet oder erst erzeugt oder bereits gestoppt wurde. Diese Lösung ist ideal und sollte gewählt werden, da so vor dem Container-Start die Abgabe zum Dateisystem und nach Container-Ende das Bewertungsergebnis vom Dateisystem kopiert werden kann, ohne die Isolation aufzubrechen. (Siehe [Inc18])

5.3. Integration in Grappa

Erweiterungspunkt

Es existieren mehrere Stellen in der Grappa-Architektur, an welchen angesetzt werden kann, um eine Auslagerung der Bewertung in einen Container zu ermöglichen.

Erweiterung vorhandener BackendPlugins Dies macht die Anpassung von Bestandscode eines jeden BackendPlugins erforderlich und ist damit keine graderunspezifische Lösung und aufwendig.

Integration im Grappa-Kern Eine direkte Integration im Grappa-Kern ermöglicht eine graderunspezifische Lösung. Gleichzeitig wird Grappa so jedoch um Funktionalität erweitert, welche vom Benutzer/Administrator ggf. gar nicht gewünscht ist, aber die Konfiguration von Grappa komplexer macht. Es besteht zudem die Gefahr, dass neue Fehler Einzug in den Grappa-Kern erhalten.

Integration durch ein neues BackendPlugin Hier wird die modulare Architektur von Grappa optimal genutzt und es handelt sich um eine graderunspezifische Lösung. Ein Benutzer/Administrator kann die Bewertung im Container so auf gewohnte Art und Weise als BackendPlugin in Grappa einbinden und konfigurieren.

Die Integration durch ein neues BackendPlugin ist aufgrund der Entwicklung eines unabhängigen Moduls mit standardisiertem Konfigurationsweg die beste Wahl.

Anstoß der Bewertung im Container

Bisher erfolgt der Anstoß einer Bewertung über das Grader-BackendPlugin, welches wiederum direkt den Grader aufruft. Der Anstoß der Bewertung muss nun allerdings im Container geschehen. Dies macht in jedem Fall eine zusätzliche Software notwendig, welche im Container residiert und den Grader aufruft.
Eine Möglichkeit ist es, für jeden Grader ein zusätzliches Skript oder Programm bereitzustellen, welches den Grader im Container direkt aufruft. Der Nachteil dieses Verfahrens ist, dass so der Aufwand zur Einbettung eines Graders in die Ausführungsumgebung sehr hoch ist, da zunächst ein entsprechendes Skript/Programm entwickelt werden muss. Weiterhin ist diese Lösung weniger graderunspezifisch.
Da bestehende BackendPlugins bereits den Aufruf eines Graders kapseln, bietet es sich an, auch BackendPlugins direkt im Container auszuführen. Somit ist nur eine einmalige Entwicklung von Software erforderlich, um beliebige BackendPlugins losgelöst von Grappa auszuführen. Diese Lösung ist graderunspezifisch und ermöglicht die Verwendung bestehender BackendPlugins sowie die aus Grappa bekannte Konfiguration. Sie sollte daher umgesetzt werden.

Dateiformat zum Austausch von Abgabe und Bewertungsergebnis

Das Übermitteln von Abgabe und Bewertungsergebnis soll, wie in Kapitel 5.2 beschrieben, über das Kopieren von Dateien erfolgen. Aus diesem Grund gilt es, ein geeignetes Dateiformat zu finden.
Möglichkeiten sind hier programmiersprachenunabhängige Formate wie XML oder JSON sowie die direkte Serialisierung in eine Byte-Folge. Da sowohl Grappa, als auch ein im Container residierendes BackendPlugin in Java geschrieben sind, kann eine direkte Serialisierung und Deserialisierung der Submission- und GradingResult-Objekte mit den von Java angebotenen Mechanismen erfolgen. XML und JSON könnten ebenfalls eingesetzt werden. Der Bestandscode würde jedoch komplexer, da bestehende Klassen häufig mit

umfangreichen Informationen für die Konvertierung angereichert werden müssen und ggf. zusätzliche Bibliotheken benötigt werden.
Sollte es später zu Performance-Engpässen bei der Bewertung kommen, kann ggf. eine anderes Dateiformat und ein anderer Serialisierungsmechanismus mit höherer Geschwindigkeit zum Einsatz kommen.

Parallelverarbeitung mehrerer Bewertungsprozesse

Die Parallelverarbeitung mehrerer Bewertungsprozesse gestaltet sich durch die Benutzung von Docker und dem damit verbundenen schnellen Start sowie niedrigen Ressourcenverbrauchs eines Containers als unproblematisch. Sollten mehrere Bewertungsanfragen gleichzeitig an das neue BackendPlugin gestellt werden, kann dieses die Erstellung von separaten Container-Instanzen vom Docker Daemon anfordern. Jede Bewertung wird anschließend parallel in der eigenen Container-Instanz durchgeführt und Ergebnisse können jeweils nach Ende einer Bewertung abgeholt werden. Um eine Überlastung des Container-Hosts zu verhindern, sollte die maximale Anzahl parallel ausführbarer Container-Instanzen konfigurierbar sein.

Nutzung mehrerer Grader

Für jeden angebundenen Grader ist bisher eine separate Grappa-Instanz erforderlich (siehe Kapitel 3.1). Hieraus folgt, dass für jeden Grader das neue BackendPlugin jeweils einmal einkonfiguriert werden muss und ein Docker Image zur Instanziierung von Containern für den entsprechenden Grader hinterlegt werden muss. Zudem wäre ein gemeinsamer Container-Host für alle Grader eine naheliegende Lösung mit wenig Overhead. Sie besitzt jedoch folgende Nachteile:

- Bei der Ressourcenzuteilung für einen Grader muss der Ressourcenverbrauch anderer Grader bedacht werden, um zu verhindern, dass ein Grader durch hohe Beanspruchung des Container-Hosts, Einfluss auf die gleichzeitig stattfindende Bewertung anderer Grader hat.
- Bei einem manuellen oder automatischen Update des Container-Hosts wäre ein Grader A ggf. erst wieder verfügbar, wenn die Komponenten eines Graders B aktualisiert wurden, obwohl A und B unabhängig sind.

Um diese beschriebenen Abhängigkeiten zwischen den eigentlich voneinander unabhängigen Gradern zu verhindern und eine höhere Verfügbarkeit einzelner Grader zu gewährleisten, ist ein separater Container-Host pro Grader die bessere Wahl, auch wenn dies mehr Overhead verursacht.
Da ein Docker Daemon jedoch in der Lage ist, mehrere Images gleichzeitig zu verwalten und zu instantiieren, ist die gewünschte Flexibilität gegeben, um auch Container-Instanzen verschiedener Grader in einem Container-Host zu administrieren.

Sollte zukünftig eine Grappa-Instanz in der Lage sein, mehrere Grader zu bedienen, könnte ein einziges BackendPlugin Bewertungen nebenläufig an die zugehörigen Container-Hosts delegieren. Dies setzt voraus, dass zur Laufzeit des BackendPlugins der Ziel-Grader aus der Submission ausgelesen und dem zugehörigen Docker Daemon zugeordnet werden kann.

6. Umsetzung

Im vorherigen Kapitel wurden Docker-Container, welche innerhalb einer von Vagrant verwalteten virtuelle Maschine residieren, als Technologie zur Umsetzung der Isolationsanforderungen ausgewählt. Weiterhin wurde sich für die Implementierung eines neuen BackendPlugins entschieden, um die Integration in Grappa zu realisieren. Innerhalb von Containern sollen bestehende Grader-BackendPlugins zum Einsatz kommen, um den Aufwand beim Einbetten von Gradern in Ausführungsumgebung gering zu halten.
Im Folgenden werden Details der Umsetzung näher erläutert.

6.1. Konzepte

Abbildung 6.1 zeigt die Gesamtarchitektur und beteiligte Komponenten zum Betreiben eines Graders in der Ausführungsumgebung unter der Annahme, dass ein Container-Host pro Grader gewünscht ist. Bereits von Grappa oder von einem Grader bereitgestellte Komponente sind blau markiert. Komponenten, welche zu Vagrant gehören, besitzen eine grüne und Komponenten, welche zu Docker gehören, eine lila Markierung. Zusätzlich entwickelte Komponenten und Konfigurationen sind rot markiert und werden nun vorgestellt.

6.1.1. Remote-Backend-Plugin

Das Remote-Backend-Plugin hat die allgemeine Aufgabe aus Grappa heraus Bewertungen in der abgesicherten Ausführungsumgebung anzustoßen und Ergebnisse aus dieser abzuholen. Die konkreten Funktionalitäten hierzu sind:

- Den Container-Host mit Vagrant erstellen, starten sowie automatisch aktualisieren und den Log abgreifen (sofern alles nicht bereits durch eine andere Instanz oder manuell geschieht).
- Container-Instanzen mit Docker erstellen, starten, überwachen und löschen.
- Ein serialisiertes Submission und GradingResult-Objekt an das Dateisystem einer Container-Instanz übermitteln.

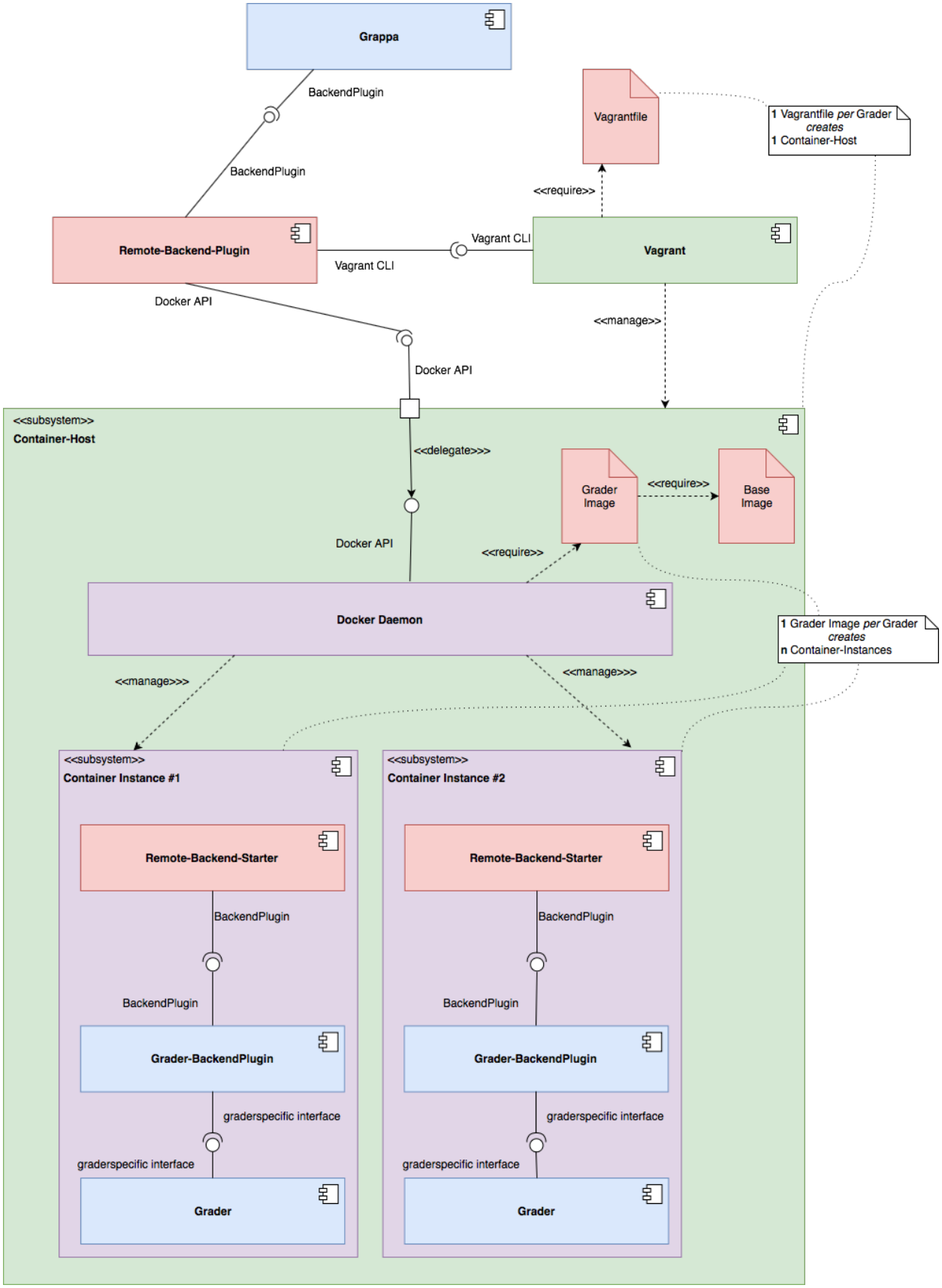


Abbildung 6.1.: Gesamtarchitektur der Ausführungsumgebung für einen Grader

- Einen Schlüssel zum Signieren eines Bewertungsergebnisses generieren und an das Dateisystem einer Container-Instanz übermitteln, um später eine zusätzliche Sicherung der Integrität des Bewertungsergebnisses zu erhalten.
- Serialisierte GradingResult-Objekte vom Dateisystem einer Container-Instanz abholen, deserialisieren und verifizieren.
- Den Log einer Container-Instanz auslesen.
- Konfigurationen bzgl. Verwaltung des Container-Hosts und Erreichbarkeit des Docker Daemons anwenden. (Siehe Kapitel 6.3.1 für Details.)
- Konfigurationen bzgl. Ressourcen- und Netzwerklimitierung für Container-Instanzen aus einer Konfigurationsdatei und aus GrdCfgs verarbeiten und gewährleisten.

Ein zugehöriges Klassendiagramm ist in Abbildung 6.2 dargestellt. ContainerLimits kapselt die Ressourcen- sowie Netwerklimitierung und ContainerGradingResult die Ergebnisse eines Container-Laufes. VmController bietet Funktionalitäten zur Steuerung des Container-Hosts und ContainerController Funktionalitäten zur Steuerung von Containern. In der Klasse RemoteBackendPlugin erfolgt die Steuerung zentraler Abläufe.
Beim Aufruf der init-Methode durch den Grappa-Kern werden mit Hilfe der Klasse VmController zunächst der Container-Host gestartet (siehe Abb. 6.3) und zukünftige Updates des Container-Hosts geplant (siehe Abb. 6.4), sofern dies in der Konfiguration aktiviert wurde. Sobald eine Abgabe bewertet werden soll, erfolgt der Aufruf der grade-Methode durch den Grappa-Kern. In dieser wird über den ContainerController ein neuer Container erstellt, die Bewertung an diesen delegiert, das Ergebnis abgeholt, der Container anschließend wieder gelöscht und das Ergebnis für den Grappa-Kern aufbereitet (siehe Abb. 6.5).
Im Falle von parallelen Bewertungsvorgängen wird die grade-Methode durch den Grappa-Kern mehrfach nebenläufig aufgerufen und über den ContainerController jeweils ein separater Container erstellt, sofern die maximale Anzahl parallel ausführbarer Container-Instanzen noch nicht überschritten wurde. Andernfalls wird mit der Container-Erstellung gewartet, bis ein Slot im Container-Host frei geworden ist.

Die Limitierung von Ressourcen eines Containers lassen sich auf Graderebene mit Parametern in der Konfigurationsdatei des Remote-Backend-Plugins definieren. Pro Programmieraufgabe lassen sich die hier festgelegten Limitierungen wiederum mit Hilfe einer GrdCfg (siehe Kapitel 3.2) überschreiben. Beides gilt jedoch nur für Container-Parameter, über die das Remote-Backend-Plugin bzw. der angesteuerte Docker Daemon eine direkte Kontrolle hat:

- Zeitlimit für einen Container
- Hauptspeicherlimit für einen Container
- CPU-Scheduler Einstellungen für einen Container
- Größenlimit des Container-Dateisystems

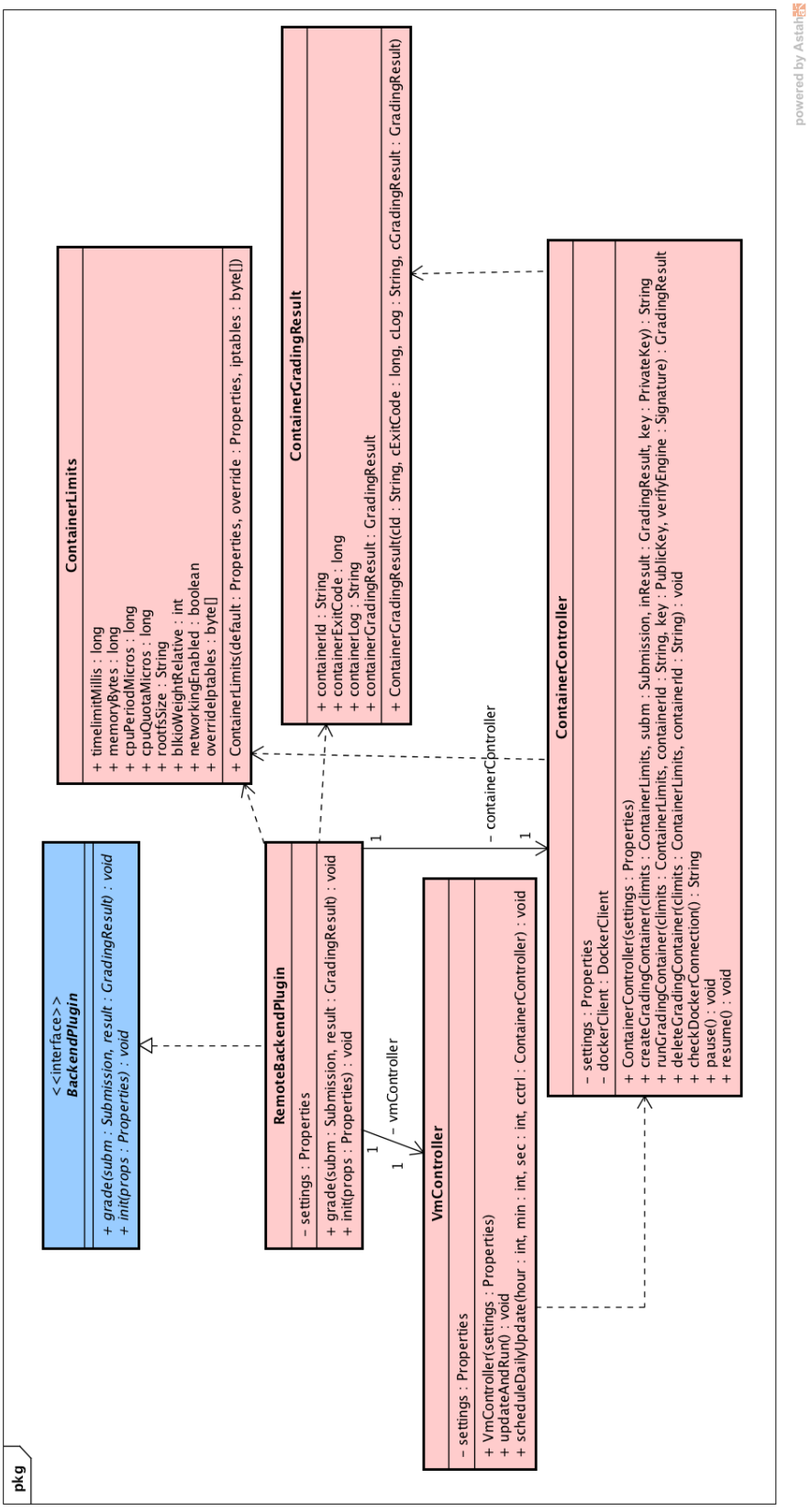


Abbildung 6.2.: Remote-Backend-Plugin Klassendiagramm

- Limitierung von I/O-Operationen eines Containers
- Aktivieren/Deaktivieren der Netzwerkschnittstelle

(Siehe Kapitel 6.3.1 für Details.)

Zusätzliche Limitierungen wie Firewall-Regeln oder Traffic Shaping können nicht von außerhalb manipuliert werden und benötigen daher Mechanismen innerhalb von Containern (siehe Kapitel 6.1.4).

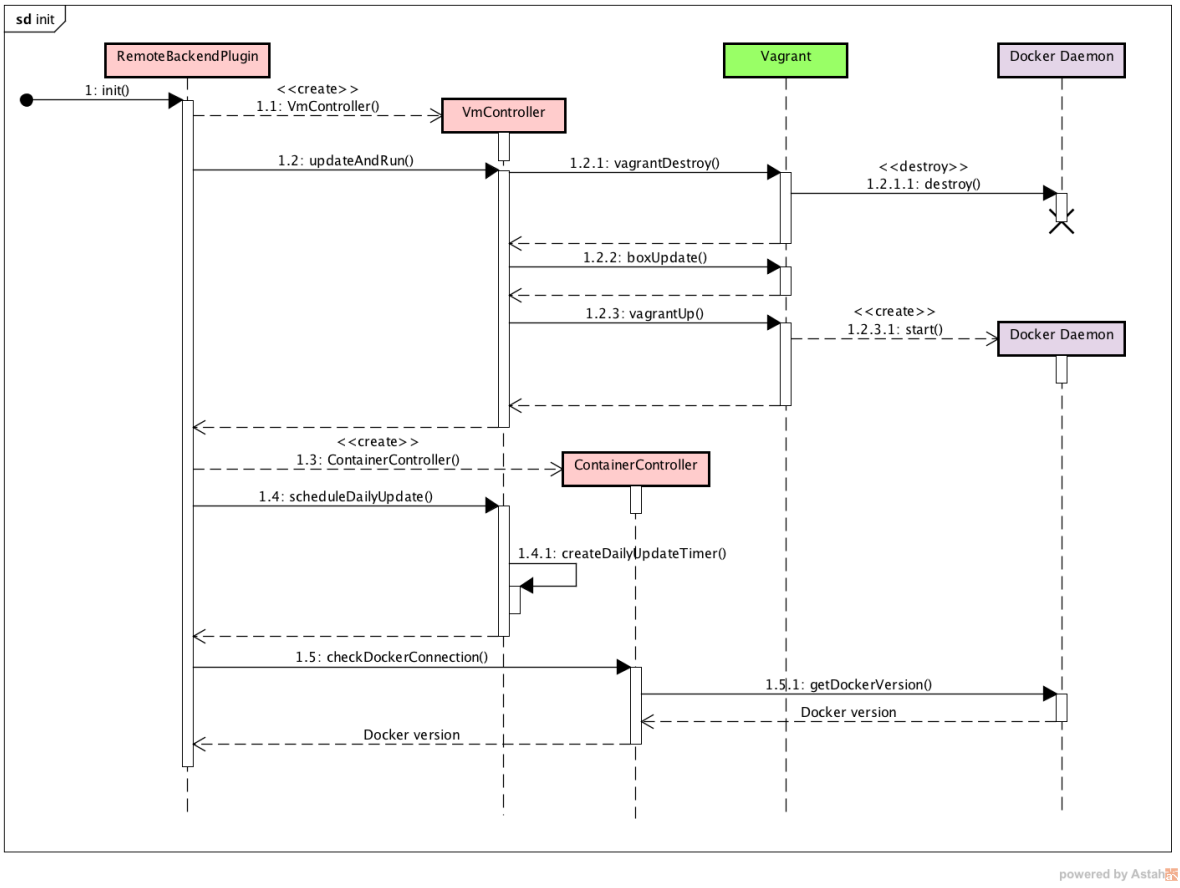


Abbildung 6.3.: Remote-Backend-Plugin init-Vorgang

6.1.2. Remote-Backend-Starter

Der Remote-Backend-Starter ist eine eigenständige Java-Applikation. Sie wird beim Start einer Container-Instanz in dieser aufgerufen, um ein beliebiges BackendPlugin von Grappa losgelöst anzustoßen. Der Remote-Backend-Starter muss hierzu folgende Aufgaben übernehmen:

- Auf dem Dateisystem befindliche serialiserte Submission- und GradingResult-Objekte sowie Schlüssel einlesen und deserialisieren.

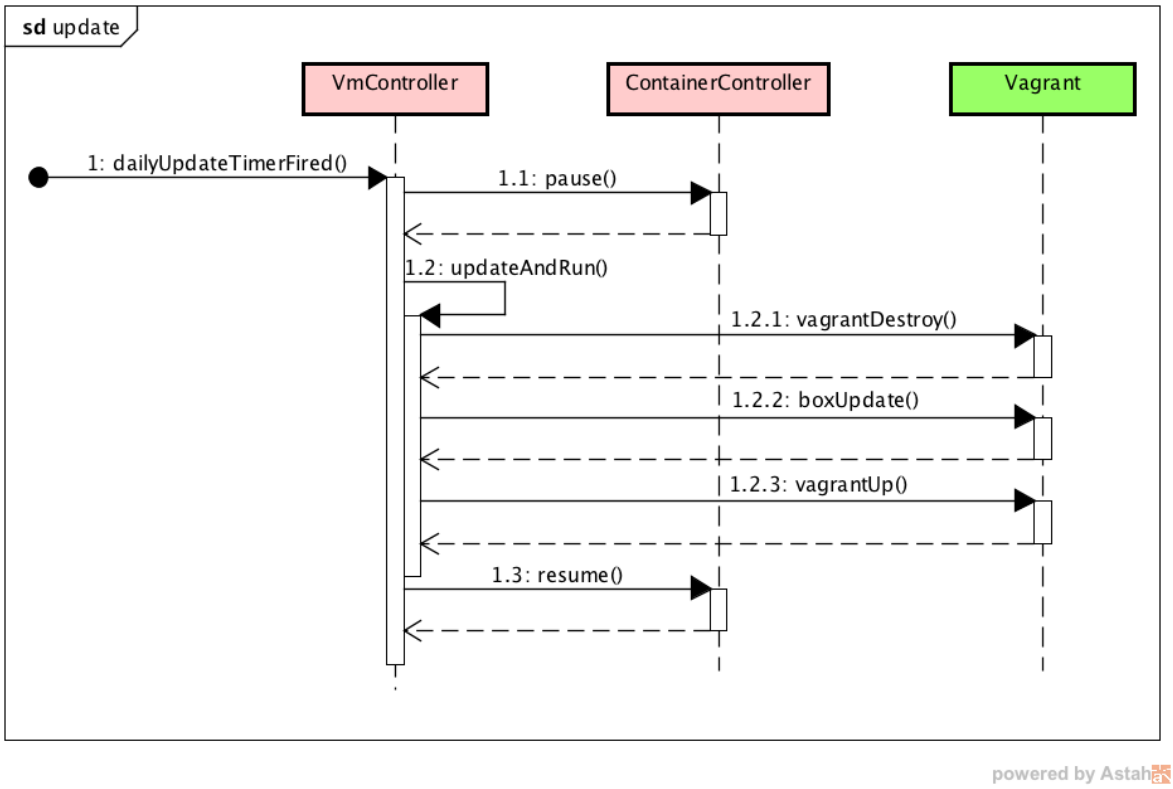


Abbildung 6.4.: Remote-Backend-Plugin update-Vorgang

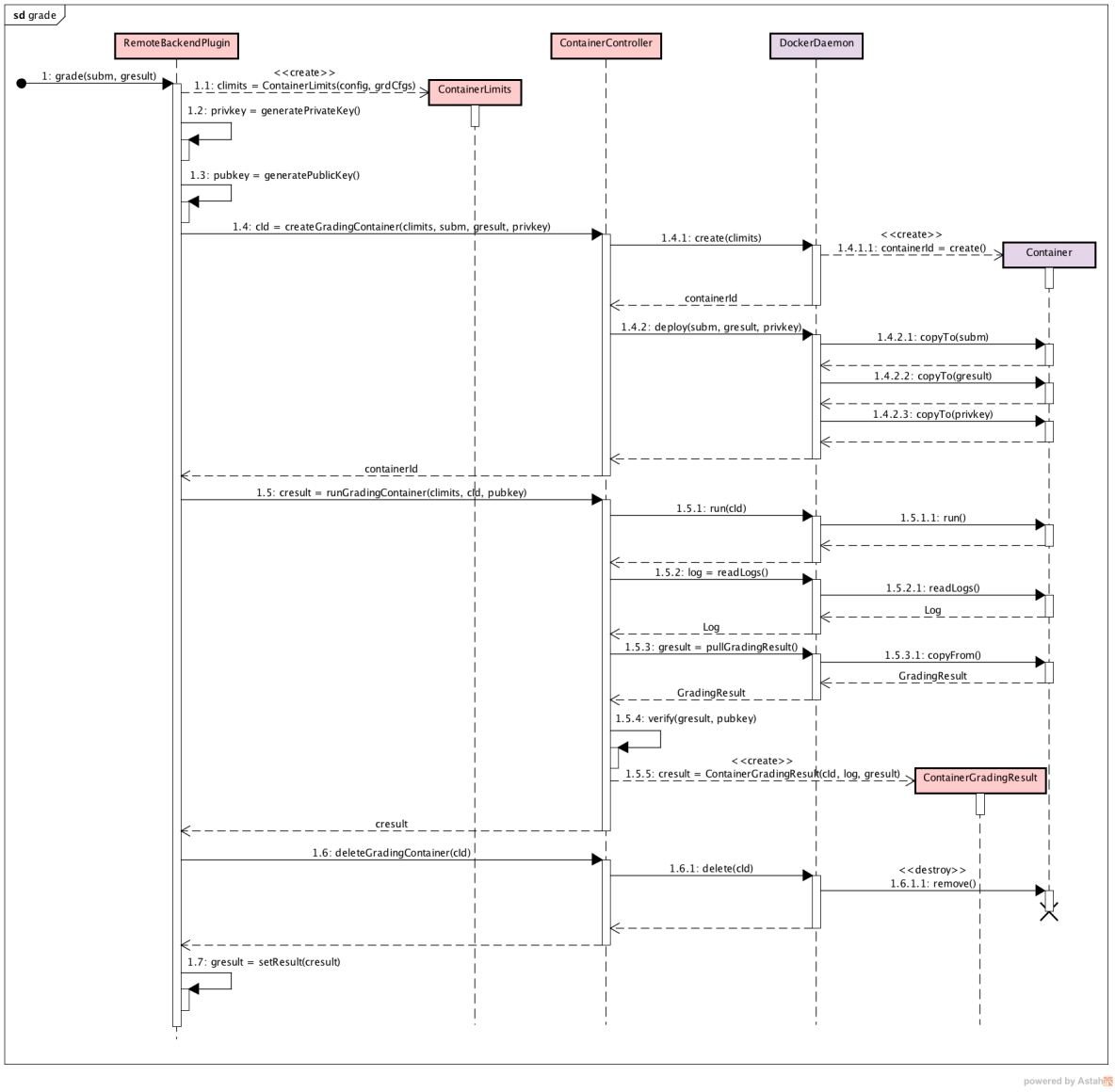


Abbildung 6.5.: Remote-Backend-Plugin grade-Vorgang

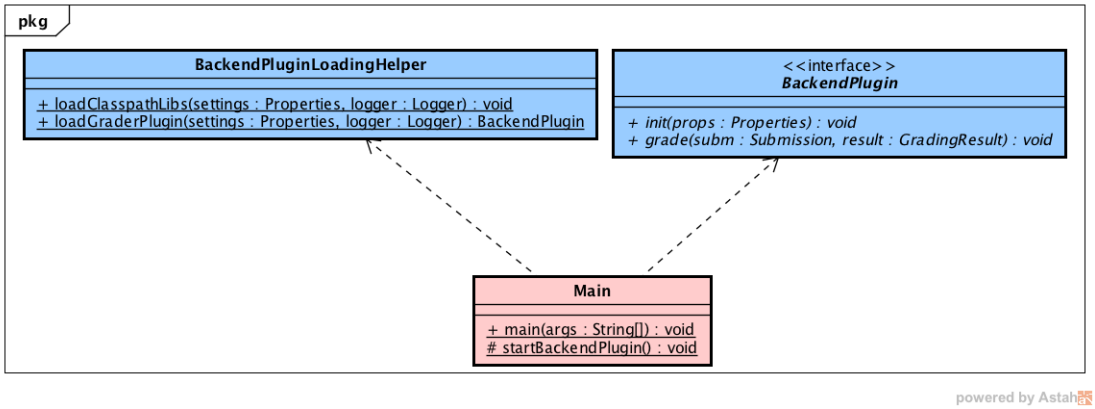


Abbildung 6.6.: Remote-Backend-Starter Klassendiagramm

- Letztere aus dem Dateisystem entfernen, um einem Missbrauch vorzubeugen.
- Das ebenfalls in der Container-Instanz hinterlegte BackendPlugin initialisieren und die Bewertung starten.
- Das Bewertungsergebnis signieren und in Form eines serialisierten GradingResult-Objektes auf das Dateisystem schreiben.

Ein zugehöriges Klassendiagramm ist in Abbildung 6.6 dargestellt. In der Klasse Main befindet sich der Eintrittspunkt der Applikation und hier findet der Anstoß der Bewertung statt. Das vom Grappa-Kern losgelöste Laden eines im Container hinterlegtem BackendPlugins erfolgt über die von Grappa bereitgestellte Hilfsklasse BackendPlugin-LoadingHelper. Diese Hilfsklasse wird ebenfalls beim Start des Grappa-Servlets zum Laden eines BackendPlugins verwendet. Der Ablauf zum Anstoßen einer Bewertung ist in Abbildung 6.7 visualisiert.
Damit der Remote-Backend-Starter beliebige BackendPlugins laden kann, werden die identischen Konfigurationsschlüssel wie in Grappa für den Ladevorgang eines Backend-Plugins innerhalb eines Containers benötigt:

grappa.plugin.grader.classpathes mit Pfaden zu Verzeichnissen und Dateien des Plugins innerhalb des Containers.

grappa.plugin.grader.fileextensions legt die zu ladenden Dateiendungen fest.

grappa.plugin.grader.class definiert die zu nutzende Plugin-Klasse.

grappa.plugin.grader.config mit dem Pfad zur Konfigurationsdatei des Plugins innerhalb des Containers.

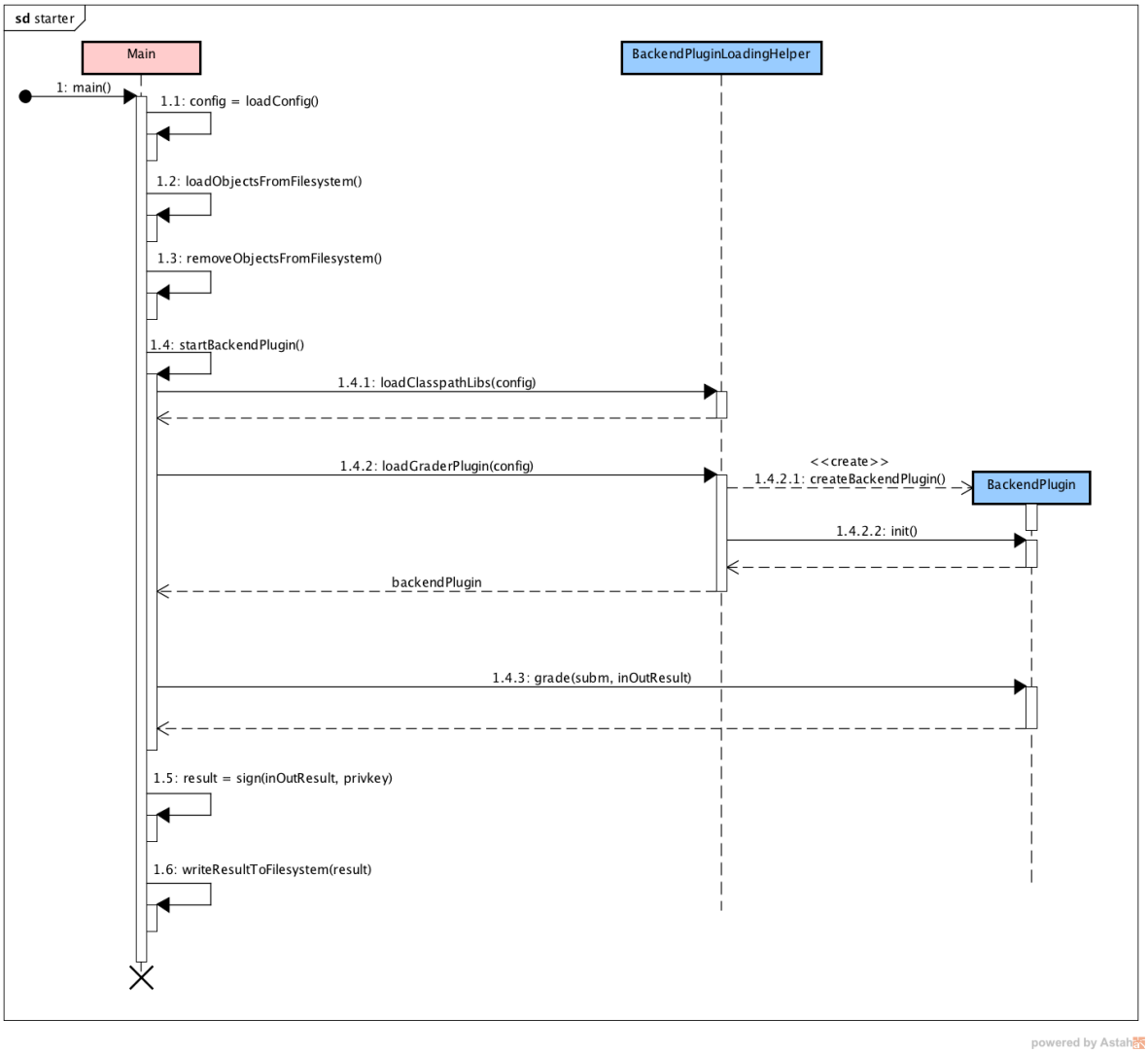


Abbildung 6.7.: Remote-Backend-Starter Bewertungsanstoß

6.1.3. Vagrantfiles

Für in die Ausführungsumgebung einzubettende Grader muss Vagrant eine virtuelle Maschine bzw. einen Container-Host erstellen können. Dies macht ein Vagrantfile notwendig, in welchem Konfiguration und Provisioning der virtuellen Maschine definiert sind. Folgendes muss im Vagrantfile enthalten sein:

- Verwendete Box (Basis-Betriebssystem)
- Name der virtuellen Maschine
- Zuteilung von Ressourcen für die virtuelle Maschine (genaue Eckdaten werden in Kapitel 6.3.1 genannt)
- Installation einer aktuellen Docker-Version
- Basis Image und Grader Image(s) für die zu erstellenden Docker-Container (siehe Kapitel 6.1.4)
- Port auf welchem die Docker API von außen erreichbar ist

Damit ein Administrator nicht direkt mit Vagrantfiles arbeiten muss, sind wichtige Parameter über eine separate config.yaml Konfigurationsdatei einstellbar. Der Aufbau von Vagrantfiles ist in Kapitel 6.2.1 beschrieben.

6.1.4. Docker Images

Um die Einbettung von Gradern in die Ausführungsumgebung so simpel wie möglich zu gestalten, existiert ein graderübergreifendes Basis Image, welches interne Komponenten der Ausführungsumgebung kapselt und ein graderspezifisches Image, welches eine konkrete Grader-Installation enthält (siehe Abb. 6.8). Eine ausführliche Beschreibung zur Erstellung und Aufbau dieser Images ist in Kapitel 6.2.2 zu finden.

Base Image

Das Base Image basiert auf einem aktuellen Ubuntu Image aus der offiziellen Docker Registry. Im Base Image ist eine für die Grader-Installation eigene Verzeichnisstruktur und ein eigener Benutzer angelegt. Somit ist neben einer einheitlichen Struktur auch eine zusätzliche Absicherung innerhalb eines Containers gegeben. Des Weiteren enthält das Image zusätzliche zur Ausführung benötigte Software wie Java zum Starten des Remote-Backend-Starters, iptables für die feine Einschränkung des Netzwerkverkehrs, iproute zum Traffic Shaping sowie den Remote-Backend-Starter selbst und ein beim Start automatisch ausgeführtes setup-Skript. Das setup-Skript besitzt die Aufgabe zunächst Restriktionen innerhalb des Containers zu gewährleisten und anschließend den Remote-Backend-Starter im Kontext eines unprivilegierten Benutzers zu starten.

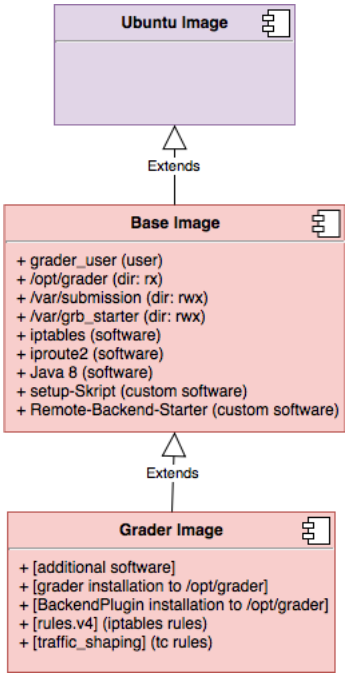


Abbildung 6.8.: Docker Image Struktur

Grader Image

Ein Grader Image erweitert das Base Image um eine konkrete Grader-Installation. Das Image kann optional die Installation zusätzlicher vom Grader benötigter Software beinhalten. Weiterhin können iptables Regeln in Form einer rules.v4 Datei und Traffic Shaping Regeln in Form einer traffic_shaping Datei für das Image hinterlegt werden, sofern eine Netzwerkverbindung benötigt wird. Das Grader Image muss zwingend ein BackendPlugin, die zugehörige Konfiguration des Remote-Backend-Starters sowie Grader-Dateien in den dafür vorgesehenen Verzeichnissen enthalten.
Neben dem Festlegen von iptables und Traffic Shaping Regeln pro Grader Image ist es auch wünschenswert, diese pro Aufgabe festlegen zu können. Aus diesem Grund ist ein Remote-Backend-Plugin in der Lage, bestehende rules.v4 und traffic_shaping Dateien nach der Container-Erzeugung mit Dateien zu ersetzen, welche in GrdCfgs hinterlegt wurden (siehe Kapitel 6.3.2 für Details).
Die Ausführungskette beim Container-Start ist in Abbildung 6.9 dargestellt.

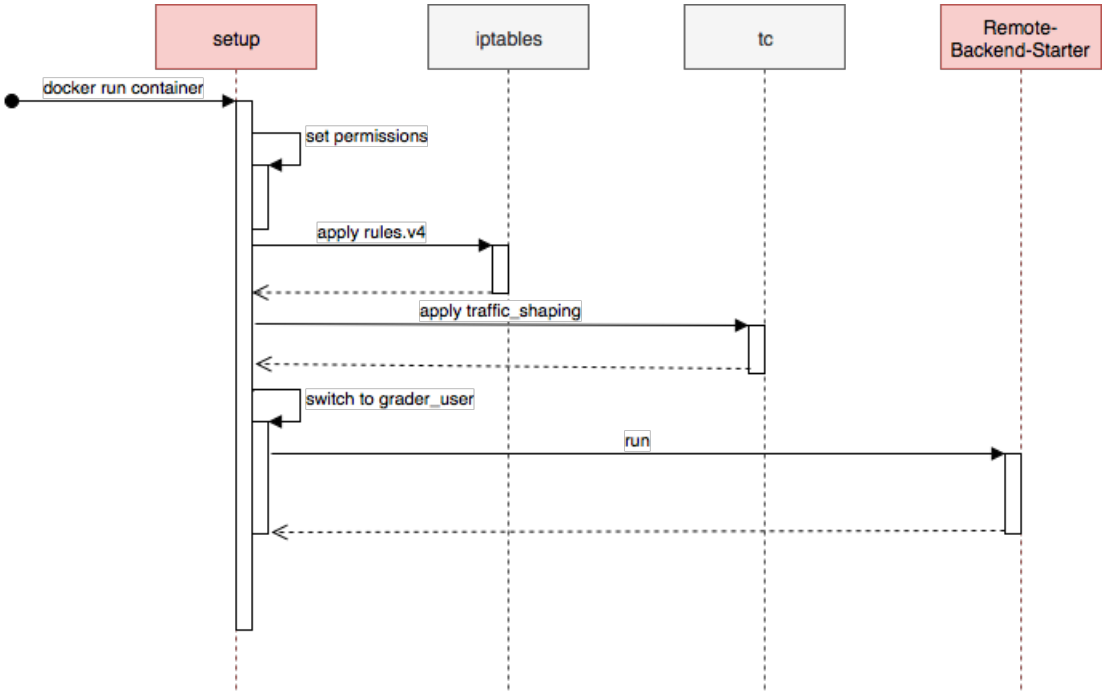


Abbildung 6.9.: Interne Ausführungskette im Container

6.2. Besondere Implementierungsdetails

6.2.1. Aufbau von Vagrantfiles für Container-Hosts

Vagrantfiles besitzen eine Ruby-Syntax. Es ist möglich, aber nicht notwendig, innerhalb eines Vagrantfiles Funktionalitäten der Ruby-Programmiersprache zu benutzen.
Ein Großteil der Konfiguration einer zu erstellenden VM erfolgt durch das simple Setzen vorgegebener Variablen innerhalb von Konfigurationsblöcken. Innerhalb des übergeordneten „config“-Blocks lässt sich beispielsweise durch Setzen der Variable „config.vm.box“ die aus der Cloud zu beziehende Basis-Box für eine VM definieren. Innerhalb eines Provider-Blocks „virtualbox“ ist z. B. mit „v.memory“ der verfügbare Arbeitsspeicher der VM definierbar, wenn die Virtualisierungssoftware VirtualBox eingesetzt wird.
Das Provisioning einer VM erfolgt durch den Aufruf eines sogenannten Provisioners innerhalb des Vagrantfiles. Mit dem Provisioner „file“ lassen sich so beispielsweise Dateien auf das Dateisystems der VM kopieren. Der Provisioner „shell“ ermöglicht z. B. die direkte Ausführung von Shell-Kommandos auf der zu erstellenden VM.
Ausführliche Details zur Entwicklung von Vagrantfiles können der Vagrant Dokumentation [Has18] entnommen werden.

Im Folgenden ist beispielhaft der Auszug des Vagrantfiles für einen Container-Hosts dargestellt und erklärt:

```
require 'yaml'

current_dir    = File.dirname(File.expand_path(__FILE__))
configs        = YAML.load_file("#{current_dir}/config.yaml")
vagrant_config = configs['configs'][configs['configs']['use']]

Vagrant.configure("2") do |config|
 config.vm.box = "ubuntu/xenial64"
 config.vm.hostname = vagrant_config['host_name']
 config.vm.synced_folder ".", "/vagrant", disabled: true

 # total vm diskspace (requires 'vagrant plugin install vagrant-disksize'
     on host)
 config.disksize.size = vagrant_config['disk_size']

 # ressources available
 config.vm.provider "virtualbox" do |v|
  v.memory = vagrant_config['memory_mb']
  v.cpus = vagrant_config['cpus']
 end

 # create xfs disk for docker (size shared among images and containers)
 config.vm.provision "shell", inline: <<-SHELL
  cd /home/vagrant
  sudo fallocate -l #{vagrant_config['docker_disk_size']} dockerdrive.img
  sudo mkfs.xfs -f dockerdrive.img
  sudo mkdir /var/lib/docker
  sudo mount -t xfs -o loop,pquota dockerdrive.img /var/lib/docker
 SHELL

 # forward docker api to host
 config.vm.network "forwarded_port", guest: 2375, host: vagrant_config['
     docker_api_port']

 # copy dockerd configuration
 config.vm.provision "file", source: "./docker", destination: "/home/
     vagrant/tmp/docker"

 # install docker-ce from official repo, load dockerd configuration and
     build docker images
 config.vm.provision "shell", inline: <<-SHELL
  cd /home/vagrant/tmp
  sudo apt-get -yq install apt-transport-https ca-certificates curl
      software-properties-common
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
      add -
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/
      linux/ubuntu xenial stable"
  sudo apt-get -yq update
  sudo apt-get -yq install docker-ce
  sudo mkdir /etc/systemd/system/docker.service.d
  sudo cp ./docker/override.conf /etc/systemd/system/docker.service.d/
```

```
      override.conf
46    sudo systemctl daemon-reload
47    sudo systemctl restart docker
48    sudo chmod +x ./docker/build-images
49    cd ./docker
50    sudo ./build-images
51    sudo rm -r /home/vagrant/tmp
52  SHELL
53  end
```

In Zeile 1 - 5 werden nötige Maßnahmen vorgenommen, um das Vagrantfile über eine separate config.yml zu parametrisieren.

In Zeile 8 - 19 erfolgen die Basiskonfigurationen der VM sowie die Zuweisung von Ressourcen.

In Zeile 22 - 28 wird für die Docker Installation ein loop device mit XFS-Dateisystem und Disk Quota Unterstützung erzeugt. Dies ist notwendig, da Docker aus technischen Gründen ein solches Dateisystem benötigt, um pro Container die Dateisystemgröße individuell einschränken zu können (siehe [Inc18]).

In Zeile 31 erfolgt die Weiterleitung des VM-lokalen Ports 2375 zu einem konfigurierbaren Host-Port, um später den Docker Daemon von Grappa (bzw. vom Remote-Backend-Plugin) aus erreichen zu können.

In Zeile 34 erfolgt das Kopieren von Dockerfiles, damit Grader Images auf der VM erstellt werden können. Des Weiteren wird eine vorgefertigte Service Konfiguration für den Docker Daemon kopiert, damit dieser auf dem VM-lokalen Port 2375 funktionsfähig ist.

In Zeile 37 - 51 erfolgt die Installation von Docker aus der offiziellen Quelle sowie die Erzeugung der Grader Images. Abschließend werden temporäre Dateien gelöscht, welche nur während der Installation benötigt wurden.

6.2.2. Aufbau von Dockerfiles zur Erstellung nötiger Docker Images

Mit Hilfe von Dockerfiles lassen sich die Inhalte von Docker Images spezifizieren, um anschließend aus einem Dockerfile ein Docker Image zu erzeugen. Es ist auch möglich, eine laufende Container-Instanz manuell anzupassen und danach diese als Docker Image zu sichern. Man erhält so jedoch nur eine relativ große Binärdatei, dessen genauer Inhalt auf den ersten Blick nicht ersichtlich ist und die Wartung entsprechend erschwert. Weiterhin gewährleistet eine regelmäßige Neuerzeugung von Docker Images mit Dockerfiles, dass sich stets aktuelle Versionen der Softwarekomponenten im Docker Image befinden. Nach der Erzeugung eines Docker Images wird dieses in einem Repository der lokalen Docker-Installation abgelegt und ist damit zur Instanziierung von Containern verfügbar.

Die Erzeugung von Docker Images geschieht im Rahmen des Provisionings des Container-Hosts (siehe vorheriges Kapitel 6.2.1). Grader Images mit den aktuellsten Softwarekomponenten sind so stets nach jedem Start und Update des Container-Hosts verfügbar.
Dockerfiles besitzen eine eigene Syntax mit eigenen Direktiven. Die wichtigsten dieser Direktiven werden nun vorgestellt:

FROM definiert ein Basis-Image, auf welchem das neu zu erzeugende Image aufbaut. FROM ist eine Pflichtinstruktion und muss genau einmal zu Beginn eines Dockerfiles benutzt werden.

ADD/COPY kopieren zur Build-Zeit Dateien oder Verzeichnisse zum Dateisystem des Images. COPY lässt sich hierbei nur auf Dateien anwenden, ADD hingegen auch auf Verzeichnisse und URLs. Diese Instruktionen sind optional und können in einem Dockerfile beliebig oft angewendet werden.

RUN erlaubt zur Build-Zeit beliebige Befehle im Image auszuführen. Beispielsweise kann mit RUN der Paketmanager aufgerufen werden, um weitere Software im Image zu installieren. RUN ist optional und kann in einem Dockerfile beliebig oft angewendet werden.

USER ermöglicht es, genau eine direkt nachfolgende Instruktion unter einem spezifizierten Benutzer auszuführen. USER ist optional und kann in einem Dockerfile beliebig oft angewendet werden. Ohne vorheriges USER Kommando wird eine Instruktion unter dem root-Benutzer ausgeführt.

CMD definiert den Standardbefehl, welcher beim Start eines Container ausgeführt wird. CMD überschreibt ggf. den Standardbefehl eines Basis-Images und sollte daher in einem Dockerfile maximal einmal verwendet werden.

Es existieren noch einige weitere Direktiven für speziellere Anwendungsfälle. Diese können der zugehörigen Dokumentation [Inc18] entnommen werden.

Dockerfile des Base Images

Im Folgenden ist ein Auszug des Dockerfiles zur Umsetzung des Base Images dargestellt und erklärt:

```
1  # Base is latest ubuntu image
2  FROM ubuntu:latest
3
4  # Add new user for grading
5  RUN ["adduser", "--disabled-password", "--gecos", "", "grader_user"]
6
7  # Create directory for grader files
8  RUN ["mkdir", "/opt/grader"]
9
10 # Create directory for submission workspace files
11 RUN ["mkdir", "/var/submission"]
```

```
12
13 # Create directories for the starter
14 RUN ["mkdir", "/var/grb_starter"]
15 RUN ["mkdir", "/var/grb_starter/in"]
16
17 ### Install software
18 RUN ["apt-get", "-yq", "update"]
19
20 # Install iptables
21 RUN ["apt-get", "-yq", "install", "iptables"]
22
23 # Install iproute2
24 RUN ["apt-get", "-yq", "install", "iproute2"]
25
26 # Install OpenJDK 8
27 RUN ["apt-get", "-yq", "install", "openjdk-8-jdk"]
28
29 ### setup skript
30 COPY ["setup",  "/setup"]
31
32 ### Add bootstrap_grader-backend script. This is implicitly called after
       container start.
33 COPY ["bootstrap_grader-backend", "/opt/grader/bootstrap_grader-backend"]
34
35 ### Add grappa-remote-backend-starter
36 ADD ["starter",  "/opt/grader/starter"]
37
38 ### End
39 # Command executed after the container is started
40 CMD ["/setup"]
```

In Zeile 2 wird als Basis-Image das aktuellste Ubuntu Image aus der offiziellen Docker Registry gesetzt.

In Zeile 3 erfolgt die Erstellung des Benutzer, unter welchem später eine Bewertung durchgeführt wird.

In Zeile 8 - 15 wird das Verzeichnis `/opt/grader` als (read-only) Installationsort für Graderdateien, das Verzeichnis `/var/submission` als Ausführungsort für Abgaben sowie das Verzeichnis `/var/grb_starter` als Ort zum Dateiaustausch für den Remote-Backend-Starter angelegt.

In Zeile 18 - 27 erfolgt die Installation von Software, welche zur Ausführung des Remote-Backend-Starters und dem Anwenden zusätzlicher Restriktionen im Container benötigt wird.

In Zeile 30 - 36 werden das initialisierende setup-Skript sowie Dateien des Remote-Backend-Starters im Image hinterlegt.

In Zeile 40 wird das setup-Skript als Startbefehl für das Base Image (und gleichzeitig für weitere Grader Images) gesetzt.

Dockerfile für das Graja Grader Image

Der Grader Graja stellt einen Installer zur Installation bereit. Da dieser auch in einem nicht interaktiven Modus ausgeführt werden kann, bietet es sich an, den Installer auch im Dockerfile zur Installation von Graja zu verwenden. Im Folgenden ist ein Auszug des Dockerfiles zur Umsetzung des Grader Images für Graja dargestellt und erklärt.

```
1  # Base is latest grappa-remote-backend-base image
2  FROM grappa-remote-backend-base:latest
3
4  ### Install grader files from install_files
5  # add graja installer
6  ADD ["install_files", "/opt/grader"]
7
8  # execute graja installer
9  RUN ["java", "-jar", "/opt/grader/installer/Graja/InstallGraja.jar", "--
       defaults", "/opt/grader/installer/installation.settings", "--unattended
       "]
10
11 # remove graja installer after installion
12 RUN ["rm", "-r", "/opt/grader/installer"]
13
14 ### Add default iptable rules
15 COPY ["rules.v4", "/rules.v4"]
16
17 ### Add traffic shaping
18 COPY ["traffic_shaping", "/traffic_shaping"]
19
20 ### remote-backend-starter config file
21 COPY ["grb_starter.properties", "/opt/grader/starter/grb_starter.
       properties"]
```

In Zeile 2 wird als Basis-Image das bereits vorgestellte Base Image verwendet.

In Zeile 6 - 12 erfolgt das Kopieren des Graja-Installers samt Plugin an den vorgesehenen Ort im Image, die Ausführung des Graja-Installers sowie anschließendes Löschen nicht mehr benötigter Dateien.

In Zeile 15 - 18 werden standardmäßige iptables und Traffic Shaping Regeln für das Grader Image hinterlegt.

In Zeile 21 erfolgt das Kopieren der Konfiguration für den Remote-Backend-Starter, damit dieser das Graja-BackendPlugin finden und initialisieren kann.

Dockerfile für das Praktomat Grader Image

Bei der Implementierung des Dockerfiles für das Praktomat Grader Image wurde sich streng an die von [Tos18] bereitgestellte Installationsanleitung gehalten. Eine Besonderheit dieses Grader Images ist, dass neben einem BackendPlugin mit Grader-Dateien

auch eine Postgres-Datenbank und ein Django-Server als Service im Image bereitstehen müssen. Da in einem Docker Container jedoch kein eigenes init-System existiert (siehe [Inc18]), ist es notwendig beide Services manuell als normale Prozesse zu starten, sofern diese während des Build-Prozesses oder für eine anstehende Bewertung benötigt werden. Im Folgenden ist ein Auszug des Dockerfiles zur Umsetzung des Grader Images für den Praktomat dargestellt und erklärt.

```
1  # Base is latest grappa-remote-backend-base image
2  FROM grappa-remote-backend-base:latest
3
4  ### Install additional software
5  RUN ["apt-get", "-yq", "update"]
6  RUN ["apt-get", "-yq", "install", "postgresql-10"]
7  RUN ["apt-get", "-yq", "install", "python2.7-dev"]
8  ...
9
10 ### Install grader files from install_files
11 # add praktomat files
12 ADD ["install_files", "/opt/grader"]
13 RUN ["mkdir", "/var/submission/data"]
14 RUN ["mv", "/opt/grader/Praktomat/SECRET_KEY", "/var/submission/data/
       SECRET_KEY"]
15
16 # install python dependencies
17 RUN ["pip", "install", "-r", "/opt/grader/Praktomat/requirements.txt"]
18 # setup db
19 USER postgres
20 RUN    /etc/init.d/postgresql start &&\
21 psql --command "CREATE USER praktomat WITH NOCREATEDB NOCREATEROLE
       NOSUPERUSER PASSWORD 'praktomat';" &&\
22 createdb -O praktomat praktomat_default &&\
23 /etc/init.d/postgresql stop
24 # populate db
25 RUN echo "local praktomat_default praktomat password" >> /etc/postgresql
       /10/main/pg_hba.conf
26 RUN su -c "/etc/init.d/postgresql start" postgres &&\
27 /opt/grader/Praktomat/Praktomat/manage-local.py collectstatic --noinput --
       link &&\
28 /opt/grader/Praktomat/Praktomat/manage-local.py migrate --noinput &&\
29 echo "from django.contrib.auth import get_user_model; User =
       get_user_model(); User.objects.create_superuser('praktomat', '
       praktomat@praktomat.com', 'praktomat')" | python /opt/grader/Praktomat/
       Praktomat/manage-local.py shell &&\
30 su -c "/etc/init.d/postgresql stop" postgres
31
32 ### Add default iptable rules
33 COPY ["rules.v4", "/rules.v4"]
34
35 ### Add traffic shaping
36 COPY ["traffic_shaping", "/traffic_shaping"]
37
38 ### remote-backend-starter config file
```

```
39 COPY ["grb_starter.properties", "/opt/grader/starter/grb_starter.
       properties"]
40
41 ### add modified setup since postgres and django server have to run before
       grading
42 COPY ["setup",   "/setup"]
```

In Zeile 2 wird als Basis-Image das bereits vorgestellte Base Image verwendet.

In Zeile 5 - 8 erfolgt die Installation von zusätzlicher Software, welche zur Ausführung des Praktomats nötig ist.

In Zeile 12 - 14 erfolgt das Kopieren der Praktomat-Dateien samt Plugin an die vorgesehenen Orte im Image.

In Zeile 17 - 31 wird die gesamte Praktomat-Umgebung inklusive der Postgres-Datenbank eingerichtet.

In Zeile 34 - 37 werden standardmäßige iptables und Traffic Shaping Regeln für das Grader Image hinterlegt.

In Zeile 40 erfolgt das Kopieren der Konfiguration für den Remote-Backend-Starter, damit dieser das Praktomat-BackendPlugin finden und initialisieren kann.

In Zeile 43 wird das vorhandene setup-Skript aus dem Base Image mit einem modifizierten ersetzt. Dies ist notwendig, da vor dem Start des Remote-Backend-Starters zunächst noch die Postgres-Datenbank und der Django-Server gestartet werden müssen.

Durch die Inklusion einer eigenen Postgres-Datenbank und einer kompletten Praktomat-Serverumgebung ist das Praktomat Grader Image etwas schwergewichtiger und der Start einer Container-Instanz mit zusätzlichen Aufwand verbunden. Eine feinere Aufteilung der Praktomat-Installation könnte hier in Zukunft Abhilfe schaffen, sodass sich nur noch für die reine Bewertung relevante Komponenten im Grader Image befinden müssen und restliche Komponenten gemeinsam verwendet werden können.

Allgemein würde eine Aufteilung der Architektur eines jeden Graders in einzelne, voneinander unabhängige Komponenten bzw. Services zu schlankeren Grader Images führen. Auch die Bewertungsdauer würde hiervon profitieren, da pro Bewertung weniger Softwarekomponenten innerhalb eines Containers initialisiert werden müssten.

6.2.3. Anpassungen am Grappa-Kern

Da es sich beim Remote-Backend-Plugin um ein eigenständiges BackendPlugin für Grappa handelt, sind keine tiefen Eingriffe in bestehenden Grappa-Code notwendig.
Das Remote-Backend-Plugin und der Remote-Backend-Starter müssen jedoch in der Lage sein, Objekte des „de.hsh.grappa.model“-Package untereinander auszutauschen. Aus

diesem Grund ist es erforderlich, dass das BaseModel zusätzlich das Serializable-Interface implementiert und Objekte des Packages eine serialVersionUID besitzen.
Um einen einheitlichen Mechanismus zum Serialisieren und Deserialisieren der Objekte anzubieten, wurde zudem die Klasse SerializationUtil im Package „de.hsh.grappa.utils“ hinzugefügt.
Weiterhin musste ein Refactoring der Klasse GrappaServlet vorgenommen werden, um den Initialisierungsprozess für BackendPlugins zu entkoppeln und diesen im Remote-Backend-Starter nutzbar zu machen. Das Resultat des Refactorings ist die gemeinsam genutzte Hilfsklasse BackendPluginLoadingHelper im Package „de.hsh.grappa.utils“ (siehe Kapitel 6.1.2).
Damit der Remote-Backend-Starter Klassen des Grappa-Kerns losgelöst von Grappa benutzen kann, wird bei einem Grappa-Build zudem zusätzlich die Bibliothek „grappa-webservice-1.1-backend-classes.jar“ erstellt, welche ausschließlich die für ein Backend relevanten Klassen enthält.

6.3. Nutzungsanleitung

6.3.1. Installation in Grappa

Die Installation des Remote-Backend-Plugins in eine Grappa-Instanz erfolgt auf die übliche Art und Weise. Der Anfang einer Grappa-Konfiguration ist hier beispielhaft dargestellt:

```
1 grappa.plugin.grader.classpathes = /etc/grappa/RemoteBackendPluginGraja/
      grappa-remote-backend-plugin-0.1.jar;
2
3 grappa.plugin.grader.fileextensions = .jar
4
5 grappa.plugin.grader.class = de.hsh.grappa.grb.plugin.RemoteBackendPlugin
6
7 grappa.plugin.grader.config = /etc/grappa/RemoteBackendPluginGraja/
      RemoteBackendPlugin.properties
8 ...
```

Sofern die Grappa-Instanz auch einen Container-Host verwalten soll, ist die Kopie eines der vorgefertigten Vagrant-Verzeichnisse mit den benötigten Grader Images erforderlich. Innerhalb der Kopie ist die Anpassung der config.yaml Datei nötig, um dem Container-Host die gewünschte Anzahl an Ressourcen zuzuteilen und die Docker API auf einem noch freien Port verfügbar zu machen. Ist eine höhere Anzahl von parallelen Bewertungen gewünscht, sollte dem Container-Host auch eine entsprechende höhere Anzahl an Ressourcen zugeteilt werden. Weiterhin sollte bei der Ressourcenzuteilung auch berücksichtigt werden, dass eine Aufgabe einem Container ggf. etwas mehr Ressourcen als im Standardfall zugestehen darf.

Eine beispielhafte config.yaml für einen ausschließlich für Graja bereitzustellenden Container-Host, auf welchem maximal drei parallele Bewertungen erfolgen sollen, ist im Folgenden dargestellt. Die Annahme ist hierbei, dass im Standardfall eine Aufgabe eine Container-Instanz mit 256 MB Arbeitsspeicher und 500 MB Speicherplatz benötigt. Diese Werte sollen aber bei Bedarf von speziellen Aufgaben noch etwas erhöht werden können.

```
---
configs:
   use: 'default'
   default:
      host_name: "XenialDockerHostGraja"
      disk_size: "20GB"
      memory_mb: 1024
      cpus: 2
      docker_disk_size: "15GB"
      docker_api_port: 2375
```

Zuletzt muss noch die Konfiguration des Remote-Backend-Plugins über die zugehörige Konfigurationsdatei definiert werden. Hier erfolgt die Spezifikation der maximalen Anzahl gleichzeitig ausführbarer Bewertungen (bzw. Containern), die Einstellung des automatischen Updates, die Festlegung der standardmäßigen Container-Limitierungen sowie die Spezifikation technischer Parameter für Vagrant und Docker. Im Folgenden ist eine Beispielkonfiguration mit allen verfügbaren Konfigurationsparametern dargestellt:

```
# Max. container instances at a time
grb.plugin.max_containers=3

# Set to true if the BackendPlugin should also manage the container host
grb.plugin.manage_container_host=true

# Set a cron like expression (minutes hours days-of-month days-of-week) to define a update intervall.
# (e.g. "0 2 * * Sun" for weekly updates on sundays 2:00 AM or "0 2 * * *" for daily updates at 2:00 AM)
# (not needed when manage_container_host=false)
grb.plugin.manage_container_host.update_intervall=0 2 * * *

# The shell to execute vagrant in (not needed when manage_container_host=false)
grb.plugin.vagrant.shell=/bin/bash

# The path to the directory where the vagrant binary resides in (not needed when manage_container_host=false)
grb.plugin.vagrant.binary.path=/usr/local/bin/

# Path to the directory where the Vagrantfile resides in to create a VM
# (not needed when manage_container_host=false)
grb.plugin.vagrant.directory_path=/etc/grappa/RemoteBackendPluginGraja/xenial_docker-host_graja

```

```
22 # URI and port to the docker daemon
23 grb.plugin.docker.uri=http://127.0.0.1:2375
24
25 # Name of the docker image to create a container from
26 grb.plugin.docker.containter_image=grappa-remote-backend-graja
27
28 # Set to true if result should be signed and checked
29 grb.plugin.require_result_signing=true
30
31 # Maximum size of container log in bytes
32 grb.plugin.max_log_bytes=5242880
33
34 # Maximum size of result in bytes
35 grb.plugin.max_result_bytes=10485760
36
37 # Timelimit after a container gets killed in milliseconds
38 grb.plugin.container.timelimit_millis=30000
39
40 # Memorylimit of a container in bytes
41 grb.plugin.container.memory_bytes=268435456
42
43 # CPU scheduler periods in microseconds
44 grb.plugin.container.cpu_period_micros=100000
45
46 # Microseconds per period before container gets throttled
47 # (Period: 100000 + Quota: 50000 = container is guranteed at most 50% of
      the CPU every second)
48 grb.plugin.container.cpu_quota_micros=70000
49
50 # Available disk space for the rootfs (e.g. 500M, 1G, etc)
51 grb.plugin.container.rootfs_size=500M
52
53 # Relative weight (between 10 and 1000) to IO block devices for container
54 grb.plugin.container.blkio_weight_relative=250
55
56 # Set to true if networking/internet should be available for the container
57 grb.plugin.container.networking_enabled=false
```

6.3.2. Aufgabenspezifische Konfiguration

Pro Programmieraufgabe lassen sich Limitierungen der Ausführungsumgebung mit GrdCfgs der Rollen `grb_limits`, `grb_iptables` und `grb_traffic_shaping` überschreiben. In einem LMS wie moodle müssen diese Rollen daher zunächst in den zugehörigen Einstellungen der Grappa-Instanz hinzugefügt werden.

Standard-Konfigurationen, welche zuvor in der Konfigurationsdatei des Remote-Backend-Plugins definiert wurden, lassen sich pro Aufgabe mit einer GrdCfg der Rolle `grb_limits` überschreiben. Es sind in der hinterlegten Datei jedoch nur Konfigurationsschlüssel

zur Limitierung von Container-Ressourcen mit dem Präfix `grb.plugin.container.` überschreibbar.

Iptables Regeln lassen sich pro Aufgabe mit einer GrdCfg der Rolle `grb_iptables` überschreiben. Dies hat jedoch nur einen Effekt, wenn für Container eine Netzwerkverbindung mit `grb.plugin.container.networking_enabled=true` aktiviert wurde. Eine exemplarische Datei mit Regeln, welche DNS und HTTP-Kommunikation erlauben, ist nachfolgend abgebildet. Eine ausführliche Dokumentation zur Definition von iptables Regeln ist in [KWM+04] zu finden.

```
1  ###################################################################
2  # iptables rules for outgoing connections in this container
3  # The container doesn't expose any ports to the outside.
4  # Therefore no need to specify any incoming rules.
5  # Last line of file must be empty.
6  ###################################################################
7  *filter
8  :INPUT ACCEPT [0:0]
9  :FORWARD ACCEPT [0:0]
10 # block all outgoing traffic
11 :OUTPUT DROP [0:0]
12 # allow loopback
13 -A OUTPUT -o lo -j ACCEPT
14 # accept related and established traffic
15 -A OUTPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
16 # accept dns
17 -A OUTPUT -p udp -m udp --dport 53 -j ACCEPT
18 -A OUTPUT -p udp -m udp --sport 53 -j ACCEPT
19 -A OUTPUT -p tcp -m tcp --dport 53 -j ACCEPT
20 -A OUTPUT -p tcp -m tcp --sport 53 -j ACCEPT
21 # accept http
22 -A OUTPUT -p tcp --dport 80 -j ACCEPT
23 # reject anything else
24 -A OUTPUT -j REJECT
25 COMMIT
26
```

Traffic Shaping Regeln lassen sich pro Aufgabe mit einer GrdCfg der Rolle `grb_traffic_shaping` überschreiben. Auch dies hat nur einen Effekt, wenn für Container eine Netzwerkverbindung aktiviert wurde. Wie sich konkrete Traffic Shaping Regeln mit dem Tool „tc“ definieren lassen, ist in [Sta01] beschrieben. Das nachfolgende Beispiel lässt sich in einer GrdCfg hinterlegen, um die Netzwerkbandbreite eines Containers auf ca. 1 MBit/s zu limitieren:

```
1 #!/bin/sh
2 tc qdisc add dev eth0 root tbf rate 1mbit burst 32kbit latency 400ms
```

6.3.3. Einbettung eines neuen Graders

Um einen neuen Grader in die abgesicherte Container-Umgebung einzubetten, ist zunächst ein umfassendes Verständnis über die Installation des Graders in eine normale Umgebung notwendig. Anschließend muss diese Installation durch Erstellung eines Dockerfiles automatisiert werden, um ein eigenes Grader Image erzeugen zu können. Zuletzt muss das neue Grader Image noch in eine Vagrant-Konfiguration eines Container-Hosts integriert werden.

Um ein Dockerfile für den neuen Grader zu entwickeln, können die in Kapitel 6.2.2 vorgestellten Docker-Konzepte und Beispiel-Dockerfiles als Referenz dienen. Ein neues Grader Image muss anschließend mindestens folgende Komponenten beinhalten:

- Das Base Image als Basis-Image
- Die Installation zusätzlich benötigter Software (sofern benötigt)
- Die Installation der Grader-Dateien nach `/opt/grader`
- Standardmäßige iptables Regeln (sofern benötigt)
- Standardmäßige Traffic Shaping Regeln (sofern benötigt)
- Die Konfigurationsdatei `/opt/grader/starter/grb_starter.properties`

Für die Integration in eine neue Vagrant-Konfiguration muss zunächst eine Kopie des Templates „xenial_docker-host_template“ angefertigt werden. Das kopierte, leere Template wird anschließend folgendermaßen befüllt:

- Kopieren eines oder mehrerer zuvor erstellten Dockerfiles inklusiver benötigter Grader-Ressourcen nach `docker/{new-Grader-name}`.
- Anpassen der Datei `docker/build-images` an vorgesehener, markierter Stelle, sodass beim Provisioning des Container-Hosts neben dem Base Image auch die hinterlegten Grader Images erstellt werden.

Nun ist die Einbettung eines neuen Graders beendet. Der neue eingebettete Grader kann, wie in Kapitel 6.3.1 beschrieben, in Grappa verwendet werden.

7. Tests und Auswertung

Im Zuge der Entwicklung von einzelnen Komponenten der Ausführungsumgebung entstanden jeweils einige Unit-Tests, um grundlegende Funktionalitäten und die Interaktion zwischen Komponenten automatisiert zu testen und sicherzustellen. Diese Tests wurden bestanden.
Im Folgenden werden die Ergebnisse von Tests vorgestellt, welche manuell in einer realitätsnahen Umgebung durchgeführt wurden. Die Umgebung umfasst:

- eine moodle-Installation
- jeweils eine Grappa-Instanz mit einkonfigurierten Remote-Backend-Plugin für die Grader Graja und Praktomat
- jeweils einen Container-Host (Eckdaten: 2 CPUs, 2048 MB RAM, 30 GB Festplatte) für die Grader Graja und Praktomat

Initialisierung/Update von Container-Hosts

Die Tabellen 7.1 und 7.2 zeigen die benötigten Zeiten zur Initialisierung bzw. zum Update von Container-Hosts bei verschiedenen Downloadraten.

Test Nr.	Graja Contaner-Host	Praktomat Container-Host
1	6min 22,345s	11min 12,036s
2	5min 18,463s	9min 22,138s
3	6min 11,528s	11min 20,427s

Tabelle 7.1.: Vagrant-Ausführungszeit bei einer Downloadrate von 1200 kB/s

Test Nr.	Graja Contaner-Host	Praktomat Container-Host
1	3min 51,276s	6min 42,011s
2	3min 46,588s	6min 42.544s
3	3min 33,742s	6min 24,679s

Tabelle 7.2.: Vagrant-Ausführungszeit bei einer Downloadrate von 3000 kB/s

Es ist zu erkennen, dass die Ausführungsdauer stark von der Downloadrate abhängig ist. Dies ist einerseits auf den Internetzugriff beim Erzeugen der VM durch Vagrant und andererseits auf den Internetzugriff beim Erstellen der Docker Images zurückzuführen. Zum Erzeugen der VM ist zunächst der Download der aktuellsten Ubuntu Vagrant Basis Box notwendig, sofern diese nicht bereits in der Vagrant-Installation vorliegt. Anschließend findet während des Provisionings, der Download der aktuellsten Docker Version statt. Das ebenfalls beim Provisioning stattfindende Erstellen der Docker Images erfordert den Download des aktuellsten Ubuntu Images als Basis sowie den Bezug aktuellster Softwarekomponenten online über den Paketmanager.

Die Laufzeit zur Erstellung eines Praktomat Container-Hosts ist länger als die eines Graja Container-Hosts, da der Praktomat deutlich mehr zusätzlich zu beziehende Software benötigt, um funktionsfähig zu sein.

Insgesamt bedeuten diese Testergebnisse, dass bei der Ausführung von Vagrant eine Bewertung für ca. 4 Minuten bzw. ca. 7 Minuten (je nach Grader bei einer Downloadrate von 3000 kB/s) vorübergehend nicht möglich ist. Zu einer Ausführung von Vagrant kommt es im Regelbetrieb genau dann, wenn ein geplantes Updates durchgeführt wird oder beim Start von Grappa festgestellt wird, dass der zu verwaltende Container-Host noch nicht verfügbar ist. Ein Herunterfahren des Container-Hosts und Grappa kann erforderlich sein, wenn Komponenten und Konfigurationen der abgesicherten Ausführungsumgebung wie das Remote-Backend-Plugin, Vagrantfiles oder Dockerfiles ausgetauscht werden müssen.
Da es sich jedoch um verhältnismäßig kurze Zeitintervalle handelt und Wartungsvorgänge nachts durchgeführt werden können, ist diese Unterbrechung des Regelbetriebs hinnehmbar. Andernfalls müsste auf die automatische Update-Funktion verzichtet werden, welches das System auf Dauer ggf. unsicherer macht oder manuellen Aktualisierungsaufwand erfordert.

Bewertungsdauer

In der Tabelle 7.3 ist die Bewertungsdauer unter Nutzung der Ausführungsumgebung dargestellt. Die Bewertungsdauer umfasst dabei die Zeit, welche zur Container-Erstellung, Ausführung der Bewertung im Container, Abholen des Ergebnisses sowie Löschung des Containers erforderlich ist.

Test Nr.	Graja	Praktomat
1	11171 ms	10421 ms
2	11166 ms	10274 ms
3	11203 ms	10330 ms

Tabelle 7.3.: Bewertungsdauer mit Ausführungsumgebung

Wie zu erkennen ist, benötigt eine Bewertung mit Graja ca. 12 Sekunden und eine Bewertung mit dem Praktomat ca. 11 Sekunden. Hierbei handelt es sich um die Regelzeiten. Sollte eine Abgabe eingereicht werden, während der zugehörige Container-Host aufgrund eines Grappa-Neustarts oder eines geplanten Updates neu gestartet wird, ist mit den zuvor beschriebenen Wartezeiten von 4 bzw. 7 Minuten zu rechnen, bevor es zu einer Bewertung kommen kann. Weiterhin besteht die Möglichkeit, dass zusätzlich auf die Beendigung vorheriger Bewertungen gewartet werden muss, sofern das Limit für die maximale Anzahl paralleler Bewertungen erreicht ist. Letzteres ist jedoch auch beim Betreiben von Grappa ohne abgesicherte Ausführungsumgebung der Fall.
Die schnellere Bewertungszeit des Praktomats, trotz der aufwendiger zu startenden Container-Instanzen, ist damit zu erklären, dass Graja ein breiteres Spektrum von Analysen auf dem eingereichten Code ausführt, während der Python-Checker des Praktomats lediglich Ausgaben vergleicht.
Die Anforderung des geringen Aufwands beim Starten von Instanzen der Ausführungsumgebung ist damit erfüllt, da im Standardfall die bisherigen Bewertungszeiten von 10 bis maximal 30 Sekunden nicht überschritten werden.

Graderunspezifische Lösung

Dass es sich bei der entwickelten Ausführungsumgebung um eine graderunspezifische Lösung handelt, in welche beliebige Grader mit geringem Aufwand eingebettet werden können, wurde bereits durch die erfolgreiche Einbettung von Graja und dem Praktomat gezeigt (siehe Kapitel 6.2.2).

Prüfung der Sicherheit

Mit Hilfe der in Kapitel 4.1 vorgestellten Use-Cases wurden Java-Test-Abgaben für Graja und Python-Test-Abgaben für den Praktomat erstellt. Um ausschließlich Restriktionen der Ausführungsumgebung und nicht der Grader zu testen, wurden graderspezifische Sicherheitsmaßnahmen deaktiviert. Die Tabelle 7.4 zeigt die Ergebnisse der Tests.

Wie zu erkennen ist, wurden alle Tests bis auf das Ausspionieren von Musterlösung bestanden. Im Fall von Graja war es möglich, eine komplette Musterlösung vom Dateisystem auszulesen und im Bewertungs-Feedback sichtbar zu machen. Beim Praktomat war es möglich, den für den Bewertungsvorgang notwendigen Unit-Test zusammen mit der erwarteten Ausgabe im Bewertungs-Feedback sichtbar zu machen.
Hieraus folgt, dass ein Grader zusätzlich selbst eine sichere Architektur bzw. eigene Sicherheitsmaßnahmen bieten muss, da es einer graderunspezifischen, abgesicherten Ausführungsumgebung im Inneren nicht möglich ist, Einfluss auf den Ablageort sensibler, interner Komponenten beliebiger Grader zu nehmen.
Weiterhin zeigen die Testergebnisse, dass mit Docker Containern eine umfassende Isolation nach außen sowie das Verhindern von DoS-Angriffen möglich ist. Auch die Fähigkeit, aufgabenspezifische Restriktionen anzuwenden, wurde bestätigt.

Test Name	Ergebnis	
	Graja	**Praktomat**
Lösung mit Endlosschleife einreichen	bestanden	bestanden
Lösung einreichen, die zu viel Hauptspeicher anfordert	bestanden	bestanden
Lösung einreichen, die zu viel Festplattenspeicher belegt	bestanden	bestanden
Lösung einreichen, die eine Netwerkverbindung benutzt	bestanden	bestanden
Lösung einreichen, die außerhalb der Ausführungsumgebung liest/schreibt	bestanden	bestanden
Lösung einreichen, die außerhalb der Ausführungsumgebung Prozesse startet/beendet	bestanden	bestanden
Lösung einreichen, die Grader-Dateien überschreibt	bestanden	bestanden
Lösung einreichen, die Berechtigungen ändert	bestanden	bestanden
Lösung einreichen, die das System herunterfährt	bestanden	bestanden
Lösung einreichen, die die Systemzeit ändert	bestanden	bestanden
Lösung einreichen, die auf sensible Teile des Hauptspeichers zugreift	bestanden	bestanden
Lösung einreichen, die eine Musterlösung ausspioniert	nicht bestanden	nicht bestanden

Tabelle 7.4.: Ergebnisse der Testfälle

8. Zusammenfassung und Ausblick

In dieser Arbeit wurde der bisherige Bewertungsablauf mit der Middleware Grappa erläutert und mögliche Angriffe auf beliebige Autobewerter vorgestellt. Es wurden die Notwendigkeit und Vorteile einer graderunspezfischen, abgesicherten Ausführungsumgebung herausgestellt und zugehörige Anforderungen definiert, um zukünftig Angriffe verhindern zu können. Anschließend fand die Evaluation diverser aktueller Technologien zur Realisierung eines Sandboxings statt. Das Ergebnis der Evaluation zeigte, dass in einer VM (= Container-Host) residierende Docker Container die derzeit beste Möglichkeit zur Umsetzung der Anforderungen sind. Für Grappa wurden dann durch die Entwicklung eines neuen BackendPlugins nötige Voraussetzungen geschaffen, um die Bewertung beliebiger Grader abgesichert in Docker Containern stattfinden zu lassen. Die exemplarische Einbettung der Grader Graja und Praktomat zeigten, dass die gewählte Lösung tatsächlich graderunabhängig und die Einbettung eines Graders in die abgesicherte Ausführungsumgebung mit wenig Aufwand zu erreichen ist. Abschließende Tests bestätigten, dass mit Docker-Containern eine umfassende Absicherung von Bewertungsvorgängen ohne größere Effizienzeinbußen möglich ist. Die Testergebnisse zeigten jedoch auch, dass auf graderspezifische Sicherheitsmaßnahmen im Inneren der Ausführungsumgebung nicht verzichtet werden kann.

Sofern Grappa zukünftig in der Lage ist, mehrere Grader aus einer Instanz zu bedienen, kann auch das Remote-Backend-Plugin angepasst werden, um mehrere Grader parallel in eigenen Ausführungsumgebungen zu bedienen. Intern müssen hierzu zur Laufzeit lediglich die Instanziierung eigener ContainerController (und VmController) pro Grader und eine entsprechende Delegation der Abgaben erfolgen.

In Zukunft kann durch das Hosten eines privaten/lokalen Vagrant Box Repositorys und einer privaten/lokalen Docker Image Registry ein erheblicher Geschwindigkeitsvorteil beim Aktualisieren und Erzeugen von Container-Hosts erzielt und damit eine höhere Verfügbarkeit gewährleistet werden. Aktuelle Container-Hosts könnten regelmäßig und automatisiert als Box exportiert und im Vagrant Box Repository abgelegt werden. Aktuelle Versionen von Base und Grader Images könnten automatisiert erstellt und direkt in der Docker Image Registry hinterlegt werden. Beide Vorgänge könnten von einem externen Tool für kontinuierliche Integration übernommen werden.
Ein Remote-Backend-Plugin muss dann lediglich die gewünschte Box aus dem Vagrant Repository beziehen und erzeugen. Die entstandene VM bezieht wiederum automatisiert vorgefertigte Grader Images aus der Docker Image Registry.

Eine durch das Remote-Backend-Plugin angestoßene Aktualisierung würde somit weniger aufwändige Provisioning- und Build-Prozesse erfordern und den Download-Aufwand für externe Ressourcen auf ein Minimum beschränken.

Sollte die Weiterentwicklung der Technologie Kata Containers eine vollständige Unterstützung aller bisherigen Docker-Funktionalitäten inklusive der Ressourcenlimitierung ermöglichen, kann in Zukunft ein einziger Docker Daemon mit Kata Containers als Container-Laufzeitumgebung für mehrere Grader verwendet werden. Container-Hosts wären durch die zusätzlich von Kata Containers gewährleistete Virtualisierung pro Container-Instanz obsolet.

A. Elektronischer Anhang

Dieser Arbeit liegt eine CD bei. Im Wurzelverzeichnis der CD ist eine zip-Datei mit den folgenden Verzeichnissen zu finden:

text/ Hier ist die Arbeit in digitaler Form abgelegt.

code/ Hier ist der Quellcode einer angepassten Grappa-Version, des Remote-Backend-Plugins und des Remote-Backend-Starters abgelegt. Außerdem sind hier die entstandenen Vagrant- und Docker-Konfigurationen zu finden. In der Datei `README.md` befindet sich eine Installationsanleitung.

tests/ Hier sind die entwickelten Test-Abgaben zur Prüfung der Sicherheit abgelegt.

Literaturverzeichnis

[BFPS17] Oliver J. Bott, Peter Fricke, Uta Priss und Michael Striewe: *Automatisierte Bewertung in der Programmierausbildung*. Waxmann Verlag, Münster, 2017. http://www.waxmann.com/buch3606.

[dj18] docker java: *Java Docker API Client*. https://github.com/docker-java/docker-java, 2018. Version: 3.0.14 Abgerufen: 2018-10-24.

[Dre12] Will Drewry: *Seccomp BPF (SECure COMPuting with filters)*. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html, 2012. Version: 4.16 Abgerufen: 2018-10-22.

[FGH+15] Peter Fricke, Robert Garmann, Felix Heine, Carsten Kleiner, Paul Reiser, Immanuel De Vere Peratoner, Sören Grzanna, Peter Wübbelt und Oliver J. Bott: *Grading mit Grappa – Ein Werkstattbericht*. In: *Proceedings of the Second Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2015)*, Wolfenbüttel, 2015. Michael Striewe. http://ceur-ws.org/Vol-1496/paper9.pdf.

[For06] Michal Forišek: *Security of programming contest systems*. In: *Information Technologies at School*, Seiten 553–563, Bratislava, 2006. https://people.ksp.sk/~misof/publications/copy/2006attacks.pdf.

[Fou18] The OpenStack Foundation: *Kata Containers documentation*. https://github.com/kata-containers/documentation, 2018. Version: 1.3 Abgerufen: 2018-10-23.

[Fri02] Stephen Friedl: *Go Directly to Jail*. http://www.linux-mag.com/id/1230/, 2002. Version: 2002-12-15 Abgerufen: 2018-10-23.

[Gar13] Robert Garmann: *Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito*. In: *Workshop „Automatische Bewertung von Programmieraufgaben“*, Hannover, 2013. http://ceur-ws.org/Vol-1067/abp2013_submission_1.pdf.

[Gar16] Robert Garmann: *Graja - Autobewerter für Java-Programme*. Technischer Bericht, Hochschule Hannover, 2016. https://serwiss.bib.hs-hannover.de/frontdoor/index/index/docId/941.

[Gar18] Robert Garmann: *Graja - Grader for java programs*. `http://graja.hs-hannover.de/`, 2018. Version: 1.10.1 Abgerufen: 2018-09-22.

[GBSO17] Florian Grummel, Ayla Brettle, David Schuster und Rainer Oechsle: *Automatische Bewertung von Python-Anwendungen*. In: *Proceedings of the Third Workshop „Automatische Bewertung von Programmieraufgaben“ (ABP 2017)*, Potsdam, 2017. Michael Striewe. `http://ceur-ws.org/Vol-2015/ABP2017_paper_04.pdf`.

[GHW15] Robert Garmann, Felix Heine und Peter Werner: *Grappa - die Spinne im Netz der Autobewerter und Lernmanagementsysteme*. In: *DeLFI 2015: Die 13. e-Learning Fachtagung Informatik*, Seiten 169–182, Bonn, 2015. Hans Pongratz, Reinhard Keil. `http://garmann.com/publications/GHW15.pdf`.

[GM84] F. T. Grampp und R. H. Morris: *The UNIX System: UNIX Operating System Security*. AT&T Bell Laboratories Technical Journal, 63(8):1649–1672, 1984. `https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1984.tb00058.x`.

[Has13] Mitchell Hashimoto: *Vagrant: Up and Running: Create and Manage Virtualized Development Environments*. O' Reilly Media, Inc., Sebastopol, 2013. `http://shop.oreilly.com/product/0636920026358.do`.

[Has18] HashiCorp: *Vagrant Documentation*. `https://www.vagrantup.com/docs/index.html`, 2018. Version: 2.2.1 Abgerufen: 2018-10-23.

[HH14] Alan Holt und Chi Yu Huang: *Embedded Operating Systems: A Practical Approach*. Springer Publishing Company, Incorporated, Heidelberg, 2014. `https://www.springer.com/de/book/9781447166030`.

[HM08] Serge E. Hallyn und Andrew G. Morgan: *Linux Capabilities: making them work*. In: *Linux Symposium*, Seiten 163–172, Ottawa, 2008. `https://www.kernel.org/doc/ols/2008/ols2008v1-pages-163-172.pdf`.

[IAKS10] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta und Otto Seppälä: *Review of Recent Systems for Automatic Assessment of Programming Assignments*. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, Seiten 86–93, New York, 2010. ACM. `http://doi.acm.org/10.1145/1930464.1930480`.

[Inc18] Docker Inc.: *Docker Documentation*. `https://docs.docker.com`, 2018. Version: 18.09 Abgerufen: 2018-10-23.

[Ini17] Open Container Initiative: *OCI Specifications*. `https://www.opencontainers.org/release-notices/v1-0-0`, 2017. Version: 1.0.0 Abgerufen: 2018-10-23.

[Ker13] Michael Kerrisk: *Anatomy of a user namespaces vulnerability*, 2013. `https://lwn.net/Articles/543273/`, Version: 2013-03-20 Abgerufen: 2018-10-23.

[KMPP07] Jim Keniston, Ananth Mavinakayanahalli, Vara Prasad und Prasanna Panchamukhi: *Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps*. In: *Proceedings of the 2007 Linux symposium*, Seiten 215–224, Ottawa, 2007. `https://landley.net/kdocs/ols/2007/ols2007v1-pages-215-224.pdf`.

[KWM+04] J. Kadlecsik, H. Welte, J. Morris, M. Boucher und R. Russell: *The netfilter/iptables project*. `https://www.netfilter.org`, 2004. Version: 4.19.4 Abgerufen: 2018-11-23.

[Lin06] Hakan Lindqvist: *Mandatory access control*. Master's Thesis in Computing Science, Umea University, Department of Computing Science, 2006. `http://lindqvist.awardspace.info/pub/MAC_exjob.pdf`.

[Lin14] Jonas Lindgren: *Analysis of requirements for an automated testing and grading assistance system*. Master's Thesis in Computer Engineering, Linköping University, Department of Computer and Information Science, 2014. `http://www.diva-portal.org/smash/get/diva2:709562/FULLTEXT01.pdf`.

[LP17] O. Liebel und Galileo Press: *Skalierbare Container-Infrastrukturen: Das Handbuch für Administratoren*. Rheinwerk Verlag, Bonn, 2017. `https://www.rheinwerk-verlag.de/skalierbare-container-infrastrukturen_4252`.

[MB12] Martin Mareš und Bernard Blackham: *A New Contest Sandbox*. In: *Olympiads in Informatics*, Seiten 100–109, Prag/Sydney, 2012. `https://www.mii.lt/olympiads_in_informatics/pdf/INFOL094.pdf`.

[Nak07] Yuichi Nakamura: *SELinux & AppArmor-comparison of secure oses*. Tokio, 2007. `https://www.elinux.org/images/3/39/SecureOS_nakamura.pdf`.

[RG05] Mendel Rosenblum und Tal Garfinkel: *Virtual machine monitors: Current technology and future trends*. Band 38, Seiten 39–47, Stanford, 2005. IEEE. `https://ieeexplore.ieee.org/abstract/document/1430630`.

[RH18] Inc. Red Hat: *rkt 1.29.0 Documentation*. `https://coreos.com/rkt/docs/latest/`, 2018. Version: 1.29.0 Abgerufen: 2018-10-23.

[SN05] James E Smith und Ravi Nair: *The architecture of virtual machines*. Band 38, Seiten 32–38, Madison, 2005. IEEE. `https://ieeexplore.ieee.org/abstract/document/1430629`.

[Spo18] Spotify: *A simple docker client for the JVM*. `https://github.com/spotify/docker-client`, 2018. Version: 8.13.1 Abgerufen: 2018-10-24.

[ŠSD15] František Špaček, Radomír Sohlich und Tomáš Dulík: *Docker as platform for assignments evaluation*. In: *Procedia Engineering*, Band 100, Seiten 1665–1671, Zlin, 2015. Elsevier BV. `https://www.sciencedirect.com/science/article/pii/S1877705815005688`.

[Sta01] Milan P Stanic: *TC–Traffic Control*. Linux QOS Control Tool, 2001. `http://arvanta.net/mps/linux-tc.pdf`.

[TB00] Albert M.C. Tam und Thomas Bader: *Linux Quota mini-HOWTO*. `http://www.linuxhaven.de/dlhp/HOWTO/mini/DE-Quota-HOWTO.html`, 2000. Version: 1.6-2 Abgerufen: 2018-10-23.

[TB10] Tocho Tochev und Tsvetan Bogdanov: *Validating the Security and Stability of the Grader for a Programming Contest System*. In: *Olympiads in Informatics*, Seiten 113–119, Sofia, 2010. `http://www.ioinformatics.org/oi/pdf/INFOL058.pdf`.

[Tho11] Bhanu P. Tholeti: *Learn about hypervisors, system virtualization, and how it works in a cloud environment*. Hypervisors, virtualization, and the cloud, 2011. `https://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/`, Version: 2011-09-22 Abgerufen: 2018-10-23.

[Tos18] Maurice Toschke: *Praktomat-Integration in Moodle*. Bachelor-Arbeit, Hochschule Hannover, Juli 2018.

[Wal07] WalkerNews.net: *Create Linux Loopback File System On Disk File*, 2007. `https://www.walkernews.net/2007/07/01/create-linux-loopback-file-system-on-disk-file/`, Version: 2017-07-01 Abgerufen: 2018-11-06.